NUMERICAL INTEGRATION WITH GRAPHICAL PROCESSING UNIT FOR QKD SIMULATION

THESIS

Virginia R. Garrett, Captain, USAF

AFIT-ENG-14-M-33

AFIT-ENG-14-M-33

NUMERICAL INTEGRATION WITH GRAPHICAL PROCESSING UNIT FOR QKD SIMULATION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Virginia R. Garrett, B.S.E.E.

Captain, USAF

March 2014

AFIT-ENG-14-M-33

NUMERICAL INTEGRATION WITH GRAPHICAL PROCESSING UNIT FOR QKD SIMULATION

Virginia R. Garrett, B.S.E.E.
Captain, USAF

Approved:

//signed//	14 Mar 2014
Douglas Hodson, PhD (Chairman)	Date
//signed//	14 Mar 2014
Michael Grimaila, PhD, CISM, CISSP (Member)	Date
//signed//	14 Mar 2014
Gary Lamont, PhD (Member)	Date

AFIT-ENG-14-M-33

Abstract

The Air Force Institute of Techology (AFIT) is developing a simulation framework to model a wide variety of existing and proposed Quantum Key Distribution (QKD) systems. This research investigates using graphical processing unit (GPU) technology to more efficiently integrate optical pulses modeled within this framework. The goal is to reduce the simulation execution time. A GPU algorithm is presented for performing numerical integration of optical pulses described by Gaussian curves to improve pulse energy and power calculations. In order to measure the performance of the algorithm a optimal timing method is needed. A timer using Comute Unified Device Architecture (CUDA) events is selected over a Windows system application programming interface (API) timer. The problem sizes studied produce speedups greater than 60x on the NVIDIA Tesla C2075 compared to the Intel i7-3610QM CPU.

I dedicate this work to my sisters for their never ending encouragement.

Table of Contents

Page

List of Figures

Figure Page

List of Tables

List of Symbols

Symbol	Definition
Symbols for Algorithms	
2^m	equally spaced abscissas
m	problem size
g	Gaussian functions within a pulse
N	level of refinement for sequential algorithm
Symbols for Tests	
n	integration calculations performed
r	replications

List of abbreviations

Abbreviation	Definition
threadIdx.x	thread index withing a block
blockDim.x	x-dimension of a thread block
gridDim.x	x-dimension of a grid
blockIdx.x	x-coordinate index of a thread block
blockIdx.y	y-coordinate index of a thread block
ID 300	ID Quantique ID 300 pulse

List of Acronyms

Acronym	Definition
AFIT	Air Force Institute of Technology
QKD	Quantum Key Distribution
OTP	One Time Pad
GPU	graphical processing unit
API	application programming interface
CUDA	Compute Unified Device Architecture
SIMD	single-instruction-stream, multiple-data-stream
SIMT	Single Instruction, Multiple Thread
FLOPS	floating point operations per second
CET	CUDA event timer

NUMERICAL INTEGRATION WITH GRAPHICAL PROCESSING UNIT FOR QKD SIMULATION

I. Introduction

CRYPTOGRAPHY, the practice and study of techniques for securing communications between two authorized parties in the presence of one or more unauthorized third parties, is the centerpiece of a centuries old battle between code maker and code breaker [4]. The strength of commonly used modern cryptographic algorithms relies on "computational security" which means the algorithms are considered secure if there is a negligible probability of discovering the key in a "reasonable" amount of time using current computational technology [5]. However, recent developments in quantum computing technology and algorithms threaten to place certain classes of commonly used classical cryptographic algorithms, such as the Ron Rivest, Adi Shamir and Leonard Adleman algorithm (RSA), at risk of being compromised [4][6].

In 1984, Bennett and Brassard proposed the first Quantum Key Distribution (QKD) protocol, BB84, to provide perfect secrecy during key distribution [7]. Using a QKD protocol, a sender and receiver create an unconditionally secure secret key by leveraging properties of quantum mechanics. QKD enables two parties to"grow" a shared secret key with any third-party eavesdropping on the key exchange introducing detectable errors. An unconditionally secure cryptosystem can be built by combining a QKD-generated key with the One Time Pad (OTP) symmetric key algorithm.

Just as in the early days of computing, each QKD system, whether commercial or research, is a unique implementation based on QKD theory and principles using currently available components, protocols, and technology. Since there are no widely accepted performance standards for evaluating QKD systems, each system designer architects their system based on their own views and needs. Because of this, the Air Force desires to model particular QKD implementations to estimate system level attributes such as effectiveness, performance, and other technical attributes.

The need to explore different QKD system implementations coupled with the limits of testing, cost of the systems, unique implementations and the relative scarcity of resources yields a problem: How does one design, develop, test, or analyse a QKD system in a resource-constrained environment? It is impractical and cost prohibitive to design, develop, build, or acquire different QKD systems for testing and evaluation purposes.

A practical alternative is to develop a simulation capability that can be used to accurately model a wide variety of existing and proposed QKD implementations for the analysis intended. At this time such a simulation capability is being designed and built at the Air Force Institute of Technology (AFIT) by a team of faculty and students.

1.1 Problem Statement

Many scientific and engineering problems are being rewritten for GPU hardware and showing significant speedup. The NVIDIA's CUDA Zone website [8] reports that some applications have shown a speedup improvement of more than two magnitudes. Programming with GPUs is based on a popular style of programming, single-instruction-stream, multiple-data-stream (SIMD). NVIDIA's GPUs are described as Single Instruction, Multiple Thread (SIMT) devices, NVIDIA's unique take on SIMD devices. An NVIDIA GPU contains many simple processors that simultaneously execute the same instruction on different data.

The CUDA *kernel* function is specified code to run on all threads during the parallel execution phase [9]. The kernel is called on the host machine and ran on the GPU device. When a kernel is called, a grid of threads is created on the device. The grid is divided into blocks and blocks are further divided into thread warps. The latest graphical processing units (GPUs) are capable of handling thousands to millions of threads simultaneously.

Currently AFIT developers report the QKD simulation require hours of wall clock time to simulate a few milliseconds of simulation time and it is believed that simulation execution time can be reduced using GPUs. However, not all problems will experience optimal performance on GPU architectures because GPUs generally lend themselves to problems that have data parallelism.

As part of the modeling effort, the shape of the electric fields for optical pulses from real-world existing laser sources is needed in order to calculate energy and power, which in turn can be used to calculate the mean number of photons present. Using the curve fitting techniques as described by [10], pulses can be described by a series of Gaussian curves. For this research, the source laser of interest is the ID Quantique ID300 and its pulse shape is presented by [11].

1.2 Approach

The integral calculation of the ID Quantique ID300 pulse for power calculations is selected to be implemented with GPU hardware and GPU programming techniques. By implementing optical pulse integration with GPU programming it is believed that the same level of accuracy and precision can be achieved while reducing execution time. GPUs use thousands to even millions of concurrent threads which allows them to process large amounts of data with very fast speed. A simplistic integration method may achieve comparable results to the current numerical integration method currently implemented in the QKD simulation. The ID300 shape presented by [11] is used to represent the ID300 pulse for numerical integration.

1.3 Research Goals and Objectives

The primary goal of this research is to determine if parallel processing with GPUs can significantly reduce the execution time of the QKD simulations.

The following objectives are identified to assist in reaching the specified research goal:

- Determine the most suitable numerical integration method for optical pulses.
- Develop a GPU numerical integration technique for optical pulses that can replace sequential integration.
- Develop an integration package using GPUs that has a small footprint and can easily be integrated into the existing QKD simulation.
- Determine at what problem size it is advantageous to use numerical integration with GPUs compared to traditional sequential programming and quantify the speedup using GPUs.
- Determine the speedup that can be achieved as the optical pulse shape definition is refined using more Gaussian shapes.
- Measure the accuracy of the GPU quadrature method as compared to sequential quadrature.

1.4 Introduction Summary

The Air Force and Department of Defense have a interest in QKD because of its impact on communications security and cryptography. GPU programming is a area in science and technology that has much growth and success. It is believed that GPU computing techniques can reduce the execution time of the QKD simulation by reducing the time spent integrating pulses. Currently simulations require hours to execute a single run. A reduction in the integration time will enhance the QKD simulations. This study attempts to determine the relative speedup of the GPU when compare to the CPU for integrating optical pulse shapes.

The following chapters present a brief description of the QKD simulation, an overview of GPU hardware and GPU programming using Compute Unified Device Architecture (CUDA) (Chapter 2). In addition, a detailed description is provided of the quadrature method chosen for this study, the trapezoid rule. The software design methodology and test methodology is presented in Chapter 3. Chapter 4 contains the results generated by testing numerical integration on GPUs and CPUs. The speedup obtained by GPU devices as the problem size grows is analyzed. The conclusion and a discussion for future research is given in Chapter 5.

II. Background

This chapter is divided into four sections. The first section gives an overview of the QKD simulation and a description of the cryptography protocol that make it possible, BB84. The QKD simulation is being developed using the OMNeT++ simulation framework. Section 2.2 gives an overview of the OMNeT++ framework and provides an understanding of the modular development within OMNeT++. The third section provides an overview of NVIDIA GPU devices and programming using the application programming interface (API) CUDA. Lastly, Section 2.4 steps through the Newton-Cotes numerical integration method, trapezoid rule, and provides a discussion of several previous application of GPU numerical integration in scientific research.

2.1 Cryptography and Quantum Key Distribution Protocol

In order to have a better understanding of the QKD simulation a basic description of cryptography and the QKD protocol is presented in Section 2.1.1 and Section 2.1.2.

2.1.1 Cryptography.

Cryptographic systems are generally composed of an algorithm for performing encryption and decryption as well as one or more keys used as parameters to lock and unlock the information [1]. Figure 2.1 shows a diagram of a simple cryptographic system. A encryption algorithm, E, transforms a plaintext message, m, into a ciphertext, $E_K(m)$, with the aid of a encryption key. The cyphertext is transformed back into plaintext using a decryption algorithm, D, and decryption key, K' [1]. An eavesdropper would have to know the decryption algorithm and secret key in order to decrypt the message. Determining the secret key is a difficult mathematical problem and gives cryptographic systems their strength. Modern computers are enabling adversaries to decode ciphertext in shorter amounts of time.

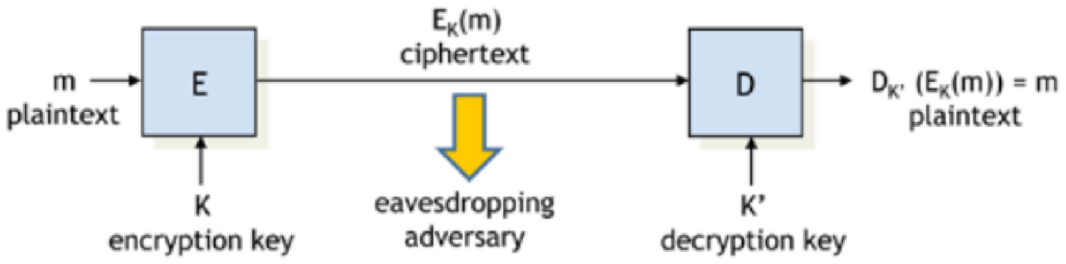


Figure 2.1: A Simple Cryptographic System [1]

OTP is an encryption algorithm that has been mathematically proven to be unconditionally secure. It requires a random generated key that is used only once. For example, parties, commonly referred to as Bob and Alice, exchange the random key. The key is equal to the length of the message to be sent. Similar to the simple cryptographic system previously described (Figure 2.1) Alice transforms the plaintext to cryptotext by Exclusive-ORing (XOR) the plaintext with the key [1]. Alice destroys the key and transmits the cryptotext to Bob. When Bob receives the cryptotext he performs an XOR operation of the cryptotext and the key; the result of the XOR operation is the plaintext message [1]. Like Alice, Bob destroys the key after performing the XOR transform. The key must be randomly generated and the keys must never be reused. Otherwise, the OTP is not unconditionally secure [1].

2.1.2 Quantum Key Distribution Protocol.

The quantum key distribution (QKD) system is based on the protocol for secure communications developed by Charles H. Bennett and and Gilles Brassard in 1984 (BB84)[7]. The BB84 protocol was the first quantum key distribution protocol and is based on the uncertainty principle of quantum mechanics. The uncertainty principle is a inherent property of all wave-like systems. In simplified terms, the uncertainty principle is the phenomenon that occurs by measuring a wave, the nature of a wave is changed. Future references to the wave will not result in the same measurement. According to Bennett and

Brassard “it is impossible in principle to eavesdrop without a high probability of disturbing the transmission in such a way as to be detected” [7].

The QKD system has two users, commonly referred to as Bob and Alice. Alice desires to send encrypted information to Bob. The QKD system is composed of a quantum and a public (non-quantum) channel [7]. Initially the quantum channel is used for exchanging random bits strings, no cyphertext is exchanged. If Alice’s and Bob’s bit strings match then the transmission over the quantum channel was not disturbed by an eavesdropper and Bob and Alice use the exchanged bits as a OTP (Section 2.1.1) to transform the plaintext to cryptotext. If an eavesdropper is detected, bit strings are discarded. Bob and Alice continue to exchange bit strings using the quantum channel until they have a long enough bit string to use as a OTP [7]. Figure 2.2 is a block diagram of the BB84 QKD system [1].

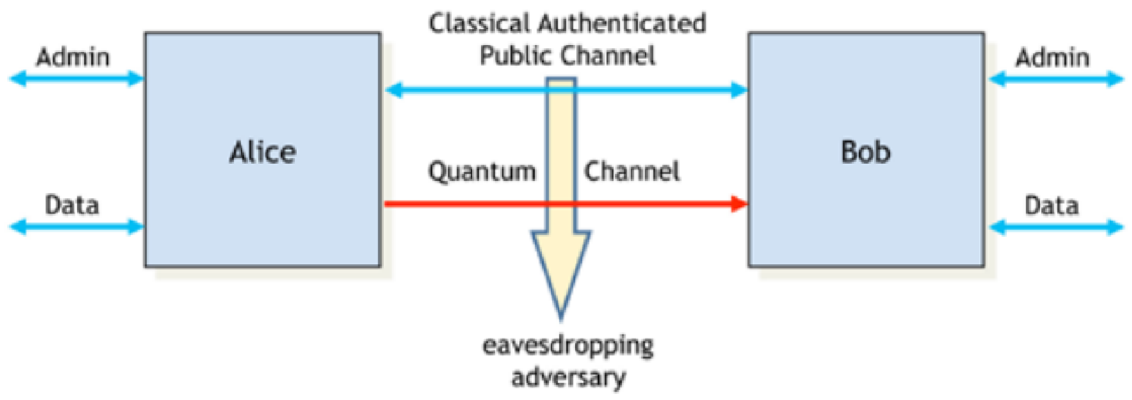


Figure 2.2: BB84 QKD System Block Diagram [1]

Alice transmits a random bit string over the quantum channel as a “train of photons.” Each photon has a random polarization basis (rectangular or diagonal) [7]. The BB84 bit basis and polarization are shown in Figure 2.3. Independent of Alice, Bob chooses a random polarization to measure each photon in the train of photons. Subsequently each photon is translated as a one or zero and Bob now has a random bit string. Bob’s bit string will be shorter than the bit string transmitted by Alice because some information is lost due to imperfect detectors and information is lost when one attempts to measure the rectilinear polarization of a diagonal polarized photon (and when the diagonal polarization is measured of a rectilinear polarized photon) [7]. Only half of Bob's bit string should be correct [7].

Secondary communication is then conducted over the pubic channel. The public channel is susceptible to eavesdropping but it is assumed the information being exchanged will not be altered if an eavesdropper exists. Using the public channel Bob and Alice determine which bits were received with the correct basis [7]. Eavesdropping will increase the amount of difference between Bob and Alice who can in turn test for eavesdroppers by comparing a subset of bits they believe to agree on over the public channel [7]. When every comparison of bits and basis between Bob and Alice agree, then they can conclude that there was no significant eavesdropping on the quantum channel [7]. The remaining bits are used as an OTP communication over the public channel.

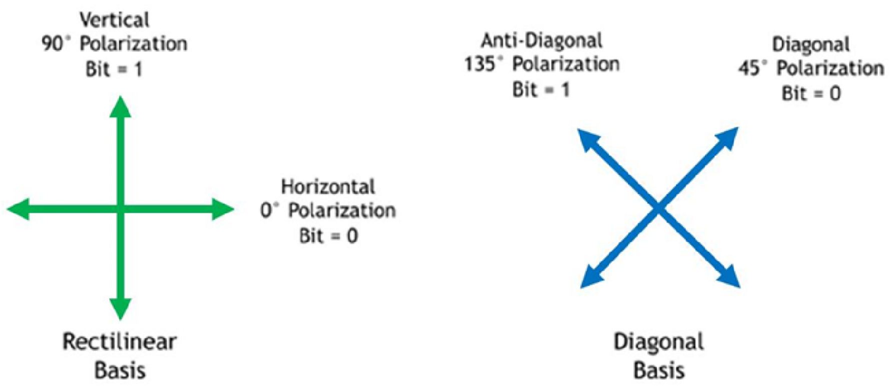


Figure 2.3: BB84 Bit Basis and Polarization [1]

2.2 OMNeT++ Discrete Event Simulation Framework

The OMNeT++ Discrete Event Simulation (DES) framework is written in C++ [2]. It is a generic and flexible architecture that can be applied to many problems. OMNeT++ was originally designed to be a an open source tool for simulation of large computer networks. It is also capable of accommodating distributed or parallel systems [2]. OMNeT++ has a modular approach that encourages reuse of code. Within OMNeT++ the definition of the model and experiments maintain a separation that allow the experiments to be changed easily while leaving the model unchanged.

A model in OMNeT++ is made up of many modules that communicate through message passing [2]. *Simple modules* are written in C++ and can be grouped into more complex modules. A grouping of simple modules in known as a *compound module*. All modules are defined as *modules types*. Previously defined module types can be used as building blocks for more complex module types. Ultimately a system module is defined from previously defined modules. The reuse of modules in other module types does not affect other users of that module type [2]. Figure 2.4 shows how modules are used in OMNeT++ [2].

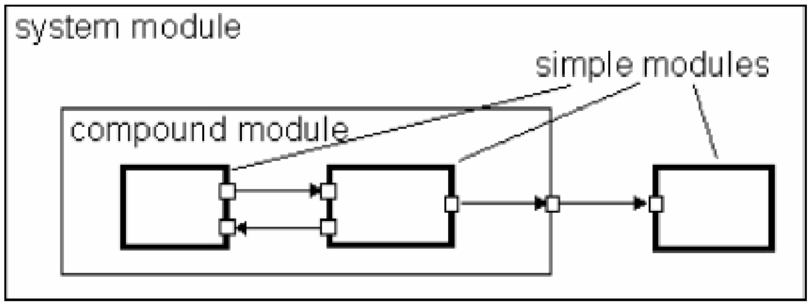


Figure 2.4: Module Structure in OMNeT++ [2]

Simple modules are the base level of all modules as they cannot be further divided. Functionality is added to simple modules by *coroutine-based programming* or *event-processing function* [2]. Coroutine-based programming launches the module code in its own thread. The thread receives control from the simulation kernel when it receives an event in the form of a message [2]. Typically coroutine-based programming runs in an infinite loop. Modules written as event-processing functions are called by the simulation kernel and given messages as input arguments. The function returns immediately after the input argument has been processed [2].

Messages are sent between modules and can contain many different types of information defined by the programmer. Messages typically travel in and out of modules via *gates*. Each module can have an input and output gate. Modules are connected at gates by defined *connections*, and connections can contain information such as propagation delays, data rate and bit error rate [2]. Lastly, modules have *parameters* which are used to pass configuration data to simple modules and define module topology [2]. Parameters are represented as objects in the program and can be passed by value or reference [2].

2.3 Graphical Processing Units

Programming with graphical processing unit (GPU) requires a different approach from the traditional sequential programming of CPUs. The two main programming paradigms for GPU programming are CUDA and OpenCL. NVIDIA released the API CUDA in 2007 which improved the ease of parallel programming with GPUs [9]. CUDA allows programmers to utilize GPU processing speeds while programming in C/C++ [9]. CUDA can be used only with NVIDIA GPU devices while the OpenCL API can be used with many GPU devices. Today's GPUs can perform many more floating point operations per second (FLOPS) than CPUs [3]. Recent releases of GPU devices have the capability of supporting double and single precision floating point operations. Figure 2.5 shows a comparison between GPU and CPU theoretical FLOPS [3].

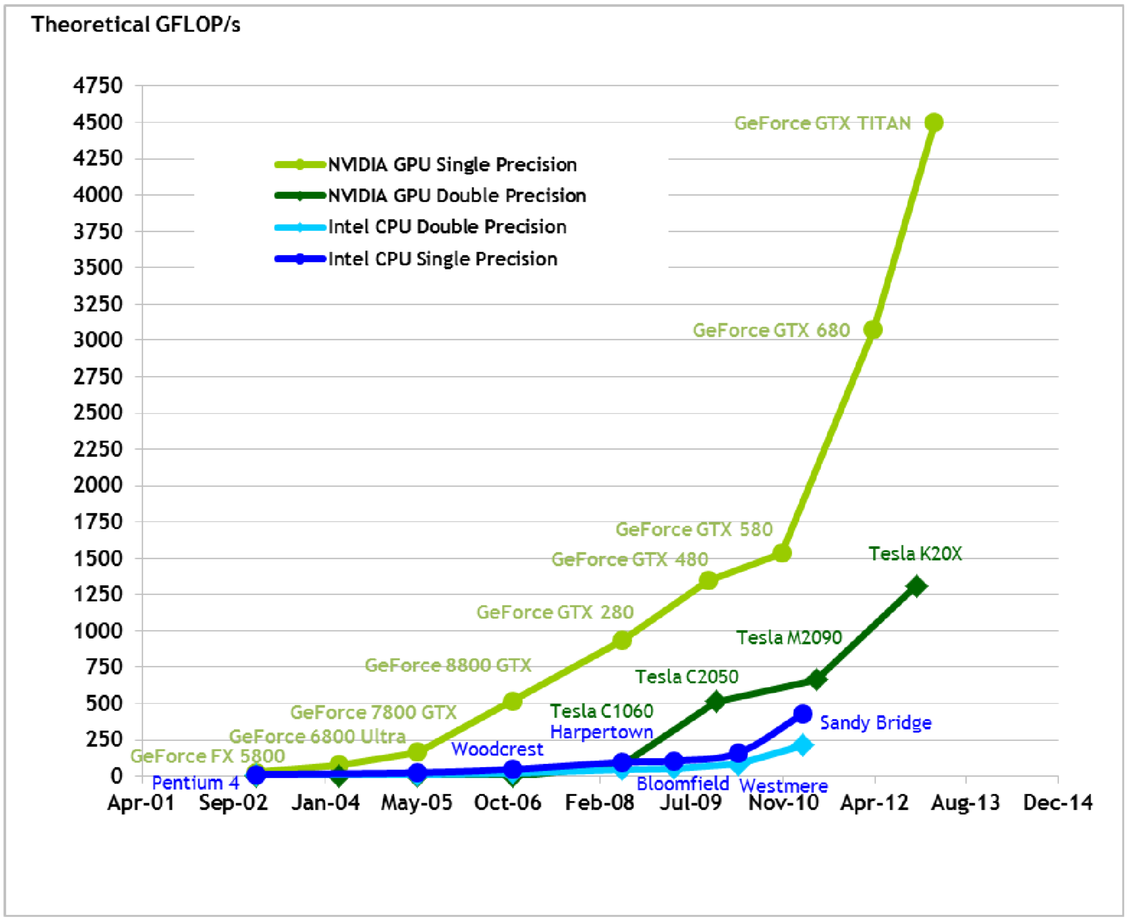


Figure 2.5: Theoretical Floating-Point Operations per Second for Intel CPUs and NVIDIA GPUs [3]

2.3.1 Many-core Systems.

GPUs are often referred to as many-core systems. Describing GPUs as many-core systems sheds some light on their architecture. The term *many* is used instead of multi, multi-core are composed of multiple CPU processors. Typically the multi-core system will have fewer than 32 processors (there are systems that have more). However, a many-core GPU system will have hundreds of GPU processors. The use of many processing cores is what gives GPUs a very high number of FLOPS.

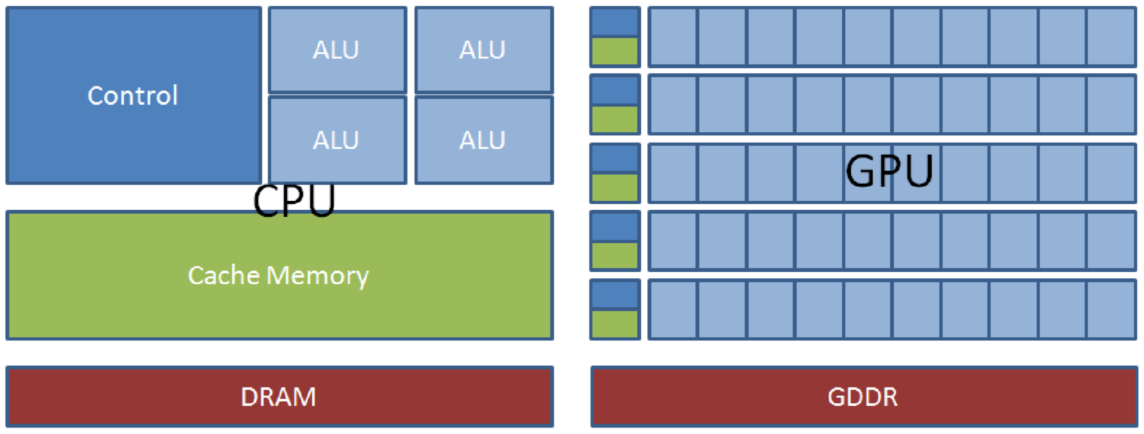


Figure 2.6: Multi-core and Many-core design philosophy [3]

GPU processors are inherently different from CPU processors. CPU processors are designed to maximize the execution speed of sequential programs [9]. Many-core systems are designed to maximize execution throughput by using many small cores [9]. The following figure (Figure 2.6) depicts the fundamental differences in multi-core and many-core systems. The GPU architecture has many transistors devoted to arithmetic operations compared to a memory cache of a tradition CPU [3].

2.3.2 SIMT Programming.

Programming with GPUs is based on a popular style of programming, single-instruction-stream, multiple-data-stream (SIMD). A SIMD machine contains many simple processors that simultaneously execute the same instruction on different data. Data is distributed across the many cores to be executed on by identical instructions. The distribution of data among cores is commonly referred to as *data parallelism*. Some problems lend themselves easily to data parallelism and SIMD programming because of the parallel nature of data within the problem.

NVIDIA's GPU programming model is closely related to SIMD programming and is refers to as Single Instruction, Multiple Thread (SIMT) [3]. Like SIMD, SIMT machines contain many-core processors. SIMT instructions control the branching and execution of a single thread. SIMT can have multiple thread paths where SIMD cannot. SIMT has the capability to handle data parallel code for coordinated threads and to handle independent thread execution [3]. SIMT is essentially a more flexible version of SIMD which allows it have a wider set of applications.

2.3.3 CUDA API.

The CUDA programming API contains functions for communicating with the GPU device. Prior to executing the many CUDA threads memory is allocated on the GPU device as needed for the problem. In addition, the required data must be copied to the GPU device. After the memory is allocated and the data has been copied to the GPU device the kernel function is executed. The *kernel* function, is the code that is ran asynchronously by all threads during the parallel execution phase [9]. If the kernel is designated with *N* CUDA threads the kernel function is executed *N* times in parallel [3]. The kernel is called on the host machine and ran on the GPU device. When a kernel is called, a grid of threads is created on the device. The grid is divided into blocks and blocks are further divided into thread warps. A thread warp is grouping of 32 threads and is the smallest number of threads that can be called by a kernel [3]. The latest graphical processing units (GPUs) are capable of handling thousands to millions of CUDA threads.

2.3.4 CUDA and NVIDIA GPU Hardware.

The NVIDIA GPU is composed of streaming multiprocessors (SMs), each SM has a number of streaming processors (SPs), also referred to as CUDA cores. Streaming processors control logic and instruction cache. For example the Tesla C2075 has 14 SMs, each with 32 SPs, that is a total of 448 SPs. The number of SPs per SM and SMs per chip varies depending the GPU. Each SM can execute hundreds to thousands of threads

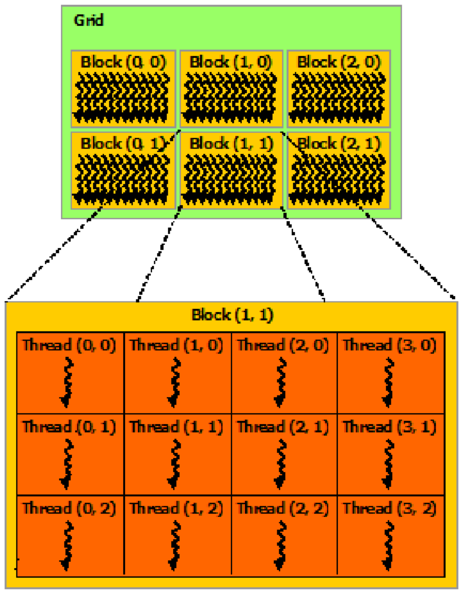


Figure 2.7: Thread, Block and Grid Structure [3]

simultaneously. The Tesla C2075 can execute 1024 threads per SP, that is 458,752 CUDA threads can be executed simultaneously on the Tesla chip.

A kernel executes a 2D *grid* of thread blocks. *Blocks* are organized into 2D or 3D arrays of CUDA threads. Blocks and threads within a block have index numbers, similar to an array, for referencing a block or thread. After a kernel has been executed, the block dimensions cannot be changed. Figure 2.7 depicts how threads are organized into blocks and grids in CUDA programming. After the grid is generated, each thread block is assigned execution resources in SMs. Up to 32 blocks can be assigned to each SM in the Tesla C2075. Care must be taken to ensure that the resources (local memory and registers) of each SM are not over allocated. Figure 2.8 depicts a kernel with eight blocks mapped to four SMs.

Figure 2.9 shows the memory structure of an NVIDIA GPU. The GPU also has *global memory* (GDDR) DRAM which is separate from the DRAM on the CPU motherboard [9].

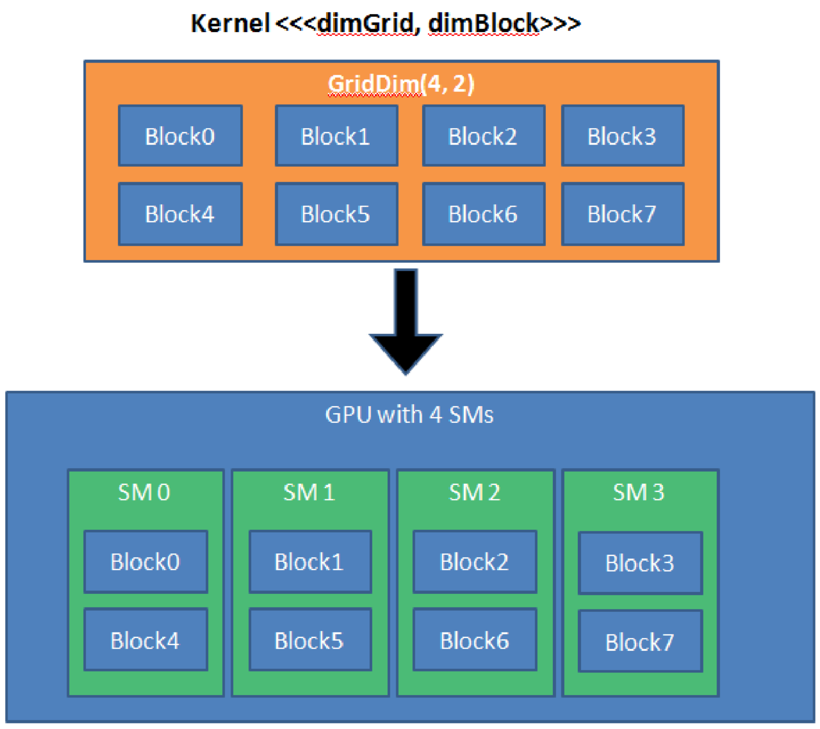


Figure 2.8: Thread Block Mapping to SM

Global memory is shared between all blocks. Every thread has the ability to read and write to global memory, however, global memory is the slowest form of device memory. In addition to GDDR many CUDA devices contain 46 kilobytes of read-only device memory, known as *constant memory*. Every thread can read from constant memory. A read from constant memory has a speed advantage over global memory because it is cached. Each read from constant memory can be broadcast to a half a warp (16 threads). Each thread block has shared memory which cannot be accessed by threads outside of the block. Lastly each thread has local memory and local registers. Shared memory and local memory only exists during kernel execution. Global memory can be accessed after a kernel has completed and can be accessed by additional kernels or copied to the CPU.

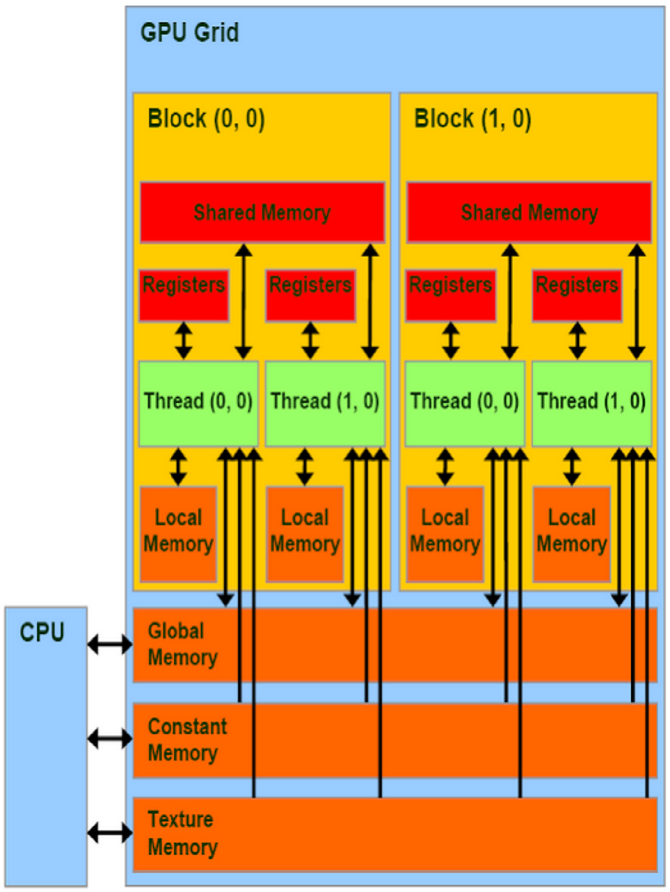


Figure 2.9: NVIDIA Hardware Structure

2.3.5 CUDA Libraries.

There are many GPU acceleration libraries that are developed for CUDA [12]. The libraries, such as cuBLAS, cuSPARSE, CUSP, and ArrayFire focus on linear algebra numerical methods. A review of CUDA libraries revealed none that contained numerical integration functions.

Unlike other parallel computing APIs, CUDA does not have a global reduction synchronization capability [13]. Reduction can be large portion of execution time for an application. There are multiple libraries that can perform a reduction or summing function, one such library is the Thrust library. Thrust is a CUDA programming API intended to make parallel programming code more concise and readable [14]. The reduction using the Thrust API is significantly faster than a sequential reduction. Thrust launches its own kernel which utilizes the shared block memory to optimize reduction.

2.4 Numerical Integration

Numerical integration, also referred to as quadrature, is widely used in many fields of study in engineering and physics. Computer simulations may perform numerous integrations throughout a single run. Classical quadrature methods are based on performing a summation of integrand values at a sequence of abscissas [1] within the range of integration [15]. The trapezoid rule is an excellent method for integrating functions of the form e^{-x^2} because it has a good convergence [15].

2.4.1 Trapezoid Method.

One of the most common techniques in simulation is the "trapezoid method" [16]. The "trapezoid method" is a closed Newton-Cotes formula which can be easily implement as a numerical method [16]. The area under a function $f(x)$ is estimated from a to b by performing a summation of trapezoids under $f(x)$. The interval $[a, b]$ is subdivided into n equally spaced intervals. Each interval has a width of $h = (b - a)/n$ where the points of evaluation are $x_0 = a$, $x_1 = a + h$, $x_2 = x_0 + h$, ... , $x_{n-1} = b - h$, $x_n = b$. Every interval forms a trapezoid and the area of all trapezoids over the interval $[a, b]$ is an approximation of the integral. As the interval width approaches zero, $h \to 0$, the number of intervals on $[a, b]$ increases $n \to \infty$, and the area under the function $f(x)$ approaches the exact solution. However, because of the limitation of computers h cannot truly approach zero and there cannot be an infinite number intervals. Thus there is a limit to the accuracy of numerical trapezoid rule implementation. The classical Newton-Cotes formula [15] for the trapezoid rule and its theoretical error [16] are written as provided in Equations 2.1-2.3. Equation

[1]Abscissas is a mathematical term for describing the horizontal coordinate of a point in a two-dimensional rectangular Cartesian coordinate system

2.1 is the composite form of the Trapezoid rule. By repeating the composite form N times the extended equation (2.2) is developed [15]. N represents the number of equally spaced sub-intervals to be calculated and $N + 1$ is the number of points to be calculated.

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(a) + f(b)) \tag{2.1}$$

$$\int_{x_0}^{x_{N+1}} f(x)dx = h\Big[\frac{1}{2}f(x_0) + f(x_1) + \cdots + f(x_{N-1}) + f(x_N) + \frac{1}{2}f(x_{N+1})\Big] + \mathrm{O}\Big(\frac{(b-a)^3 f''(\xi)}{N^2}\Big) \tag{2.2}$$

$$\in_n (\xi) = \frac{(b-a)^3}{12N^2} f''(\xi) \tag{2.3}$$

The observed error is bounded by the theoretical error, $\in_n (a) \leq Actual\ Error \leq \in_n (b)$. The true error cannot be determined beforehand, however, it is clear that the error is a function of the interval size. Reducing the interval width (and in turn using a large number of intervals) decreases the error. The number of intervals needed for a specified error tolerance can be estimated using Equation 2.4 [16].

$$n = \left\lceil \frac{(b-a)}{\sqrt{12 \in_n (\xi)}} \right\rceil \tag{2.4}$$

Classical Newton-Cotes integration requires $O(n)$ operations for a single integration. With more advanced integration techniques (i.e. Gaussian Quadrature) $O(n)$ operations are still required. However, the advanced techniques can achieve the same $\in_n (\xi)$ with far fewer intervals [15]. The “trapezoid rule” is exact for polynomial up to and including degree-1 [15]. Other Newton-Cotes formula are capable of handling higher order polynomials, such as the Simpsons rule, Simpsons 3/8 rule, and Bodes rule [15]. By leveraging the raw number of threads capable by GPUs the classical Newton-Cotes formula is capable of achieving improved execution time as compared to Newton-Cotes integration with sequential programming.

2.4.2 Previous Implementation of GPU Numerical Integration.

Many scientific and engineering problems utilize multidimensional integration [16]. As the dimentionality of the integral grows the work required to compute the integral using Newton-Cotes become $O(n^k)$ (where k is the number of dimensions). Advanced numerical integration methods such as Gaussian quadrature, and adaptive quadrature [2] can be used to reduce the number of calculates necessary to achieve the same accuracy as Newton-cotes integration [16] [15].

Adaptive quadrature requires a recursive algorithm to divide the function into subintervals to be integrated [15]. Several works utilize GPUs to perform multidimentional numerical integration [17][18][19]. Both [17] and [18] state that adaptive quadrature techniques are not suited for GPU programming because they do not take advantage of data parallelism. Nelson states that the execution time using adaptive quadrature is doubled and that a efficient method for using adaptive quadrature with GPUs is unknown [17]. For the reasons stated by [17] and [18] adaptive quadrature is not pursued as a viable method to integrate the optical pulse.

2.5 Background Summary

This chapter has provided a brief background on cryptography and describes the unconditional security that can be achieved using a one-time pad. It also describes the theory behind the BB84 protocol used in the QKD system and how the QKD system can maintain security with the eavesdroppers present. A short description is provided of the OMNeT++ simulation framework and particular attention was given to the modular design of OMNeT++. The modular design allows programmers to produce very large network simulations with ease. An overview of NVIDIA SIMT GPUs presented as well as many of the nuances of programming using the thread/block/grid structures within CUDA. Lastly,

[2]The QKD simulation developers report that the QAG adaptive integration from the GNU Scientific library is the quadrature method currently implemented in the Air Force Institute of Technology QKD simulation

the numerical integration technique, the trapezoid rule, is presented as well as research areas that have previously utilized GPUs for numerical integration.

III. Methodology

This chapter proves a description of the design and test methodology used to evaluate numerical integration of optical pulses with CUDA. The design methodology is presented in Section 3.1 which provides a description of how the ID300 pulse is modeled. In addition, a CUDA quadrature algorithm is presented to integrate the ID300 pulse. In order for the CUDA algorithm to be evaluated an appropriate timer for CUDA and sequential programming must be selected. Section 3.2 provides the search for an optimal timer. After a optimal timer is selected that timer is then used to evaluate the performance of the sequential quadrature algorithm and the CUDA based algorithm. A description of the tests used to evaluate both is also in Section 3.2.

3.1 Methodology of Design

The design methodology first presents the ID300 pulse approximation as a sum of Gaussian shapes. The sequential and CUDA algorithm is given as well as an overview of the software design of the CUDA quadrature technique.

3.1.1 Workload.

An approximation of the ID300 pulse is used for all quadrature experiments in this study. The pulse is composed of three Gaussian functions as described by [11]. The amplitude (A), mean (μ), and standard deviation (σ) of each Gaussian is provided in Equation 3.2. The integration limits used are from $a = 0$ to $b = 4e^{-10} seconds = 400$ picoseconds. The ID300 pulse is shown in Figure 3.1.

$$Pulse_{id300} = Gaussian_1 + Guassian_2 + Guassian_3 \tag{3.1}$$

$$Gaussian = A\frac{1}{\sqrt{2\pi\sigma^2}}e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{3.2}$$

$$\begin{array}{lll} A_1 = 45.1 & \mu_1 = 95.4844e^{-12} & \sigma_1 = 19.8151e^{-12} \\ A_2 = 38.064 & \mu_2 = 181.125e^{-12} & \sigma_2 = 58.1389e^{-12} \\ A_3 = 4.75752 & \mu_3 = 352.322e^{-12} & \sigma_3 = 49.0169e^{-12} \end{array}$$

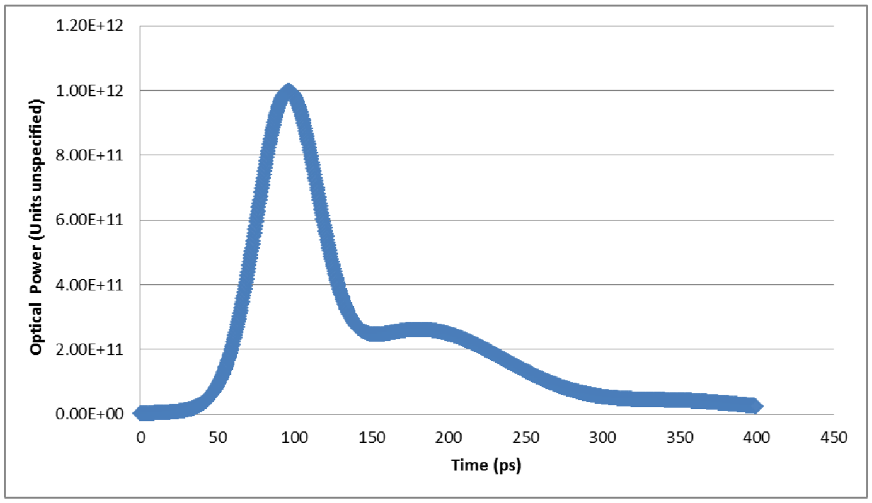


Figure 3.1: The ID Quantique ID300 pulse

3.1.2 Sequential Quadrature Algorithm.

The sequential algorithm for the trapezoid method is adapted from the trapezoid algorithm in the third edition of *Numerical Recipes* [15] and is used as the baseline for comparisons with GPU devices for calculating speedup (Equation 3.3). The sequential algorithm (Algorithm 1) is developed from the extended formula for the Trapezoidal rule (Equation 2.2). Other quadrature methods such as the Simpson-rule and Gaussian quadrature could also be implemented with CUDA. The trapezoid method is the most basic and the first step in numerical integration and is chosen in order to provide a baseline for future integration methods using CUDA.

The first iteration of the algorithm evaluates the integrand at the limits of integration, $[a, b]$. Each subsequent iteration adds 2^{N-2} points between a and b, where N is the level of refinement [15]. Figure 3.2 shows the refinement process followed by the trapezoid

Algorithm 1: Extended Trapezoid Rule, Sequential Algorithm

```
Data: $g$, $a$, $b$, $m$, and ID300 parameters
Result: Solution
for (int N=1; N<=m+1; N++) do
    if First Iteration, N=1 then
        // calculate end points
        $Solution = \frac{b-a}{2}(f(a) + f(b));$
    else
        for (k=1;k<N-1;k++) do
            $h = \frac{b-a}{2^{N-2}}$ ;
            $x = \frac{1}{2}h + a;$
            $sum = \sum_{j=1}^{2^{N-2}} f(x + (j-1)h);$
            $Solution = \frac{1}{2}(Solution + (h \times sum));$
```

algorithm by adding the results from each iteration. The integrand is evaluated a total of $2^m + 1$ times, which is equivalent to calculating the area of 2^m trapezoids. Each evaluation of the integrand $f(x)$ calculates the three Gaussian functions, $f(x) = g_1(x) + g_2(x) + g_3(x)$.

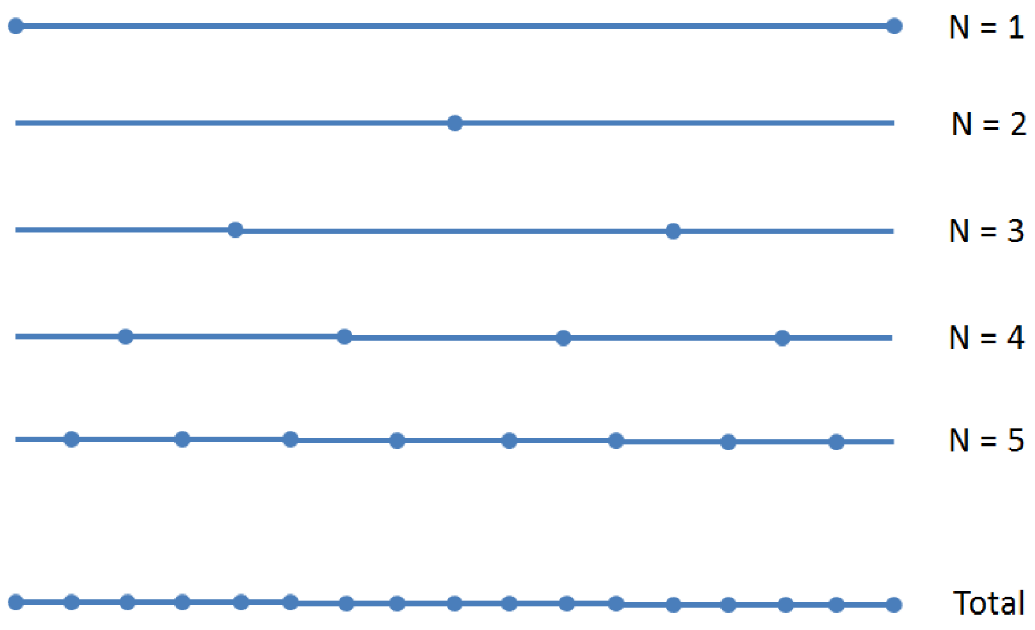


Figure 3.2: Each sequential call to the trapezoid routine evaluates unique points on the integrand. The bottom line shows total point evaluations after the 5th recursive call.

3.1.3 CUDA Quadrature Algorithm.

The CUDA quadrature algorithm (Algorithm 2) has three stages, initialization, kernel, and reduction. Prior to the kernel launch, the GPU device and problem parameters are initialized. In initialization the step size h for all threads is calculated, in addition the array Kd is allocated on the GPU device. Each element of array Kd contains the solution for one interval of the function. Lastly the initialization sets the block and grid dimensions. A one dimensional block of up to size 1024 is used. The maximum number of threads

Algorithm 2: Trapezoid Rule, CUDA Algorithm

```
Data: $g$, $a$, $b$, $m$, ID300 parameters and BlockWidth
Result: Solution
begin initialize
    // Configures the GPU device and calculate step size
    h = $\frac{b-a}{2^m}$;
    probSize = $g \times 2^m$;
    Allocate array Kd size of probSize in device memory;
    Create Thrust pointer wrapper for Kd;
    Block Dimensions = dim3(BlockWidth, 1);
    Grid Dimensions = dim3$\left(g, \frac{2^m}{BlockWidth}\right)$;
begin Launch Kernel
    // Not a loop
    foreach Thread do
        $i \leftarrow$ thread id within block;
        $z \leftarrow$ global thread id;
        $x_0 = a + (blockIdx.y \times blockDim.x \times h)$;
        $x_i = (h \times i) + x_0$ ;
        $xf = x_i + h$;
        Kd[z] = $\frac{h}{2} \times (f(x_i) + f(x_f))$;
Solution = ThrustReduce(Kd);
return Solution;
```

per block of the Tesla and Quadro GPU is 1024. If the problem size is larger than the maximum number of threads per block then the problem is divided among several blocks in the grid. The y-dimension of the grid represent the division of the domain of the problem. The x-dimension of the grid represents the specific Gaussian function of the ID300 pulse. Figure 3.4 depicts how the problem is divided among multiple blocks. The ID300 pulse contains three Guassian functions but is not limited to three Guassians. CUDA has the capability of handling pulses containing many Guassian functions. This research tests up to 15 Gaussians within a pulse.

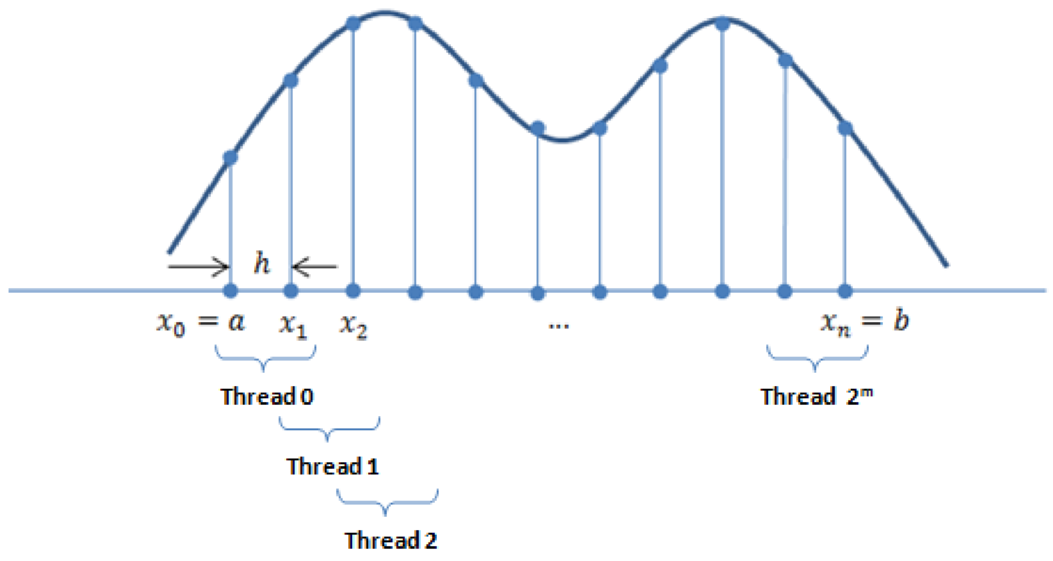


Figure 3.3: The trapezoid rule thread assignment for the CUDA quadrature method

The kernel performs the composite form of the Trapezoid rule (Equation 2.1). Each CUDA thread performs Equation 2.1 exactly once. Using this approach each internal point is evaluated twice, however, thread branching within the kernel is avoided completely. The results from each interval calculated by the trapezoidal method are stored in the device

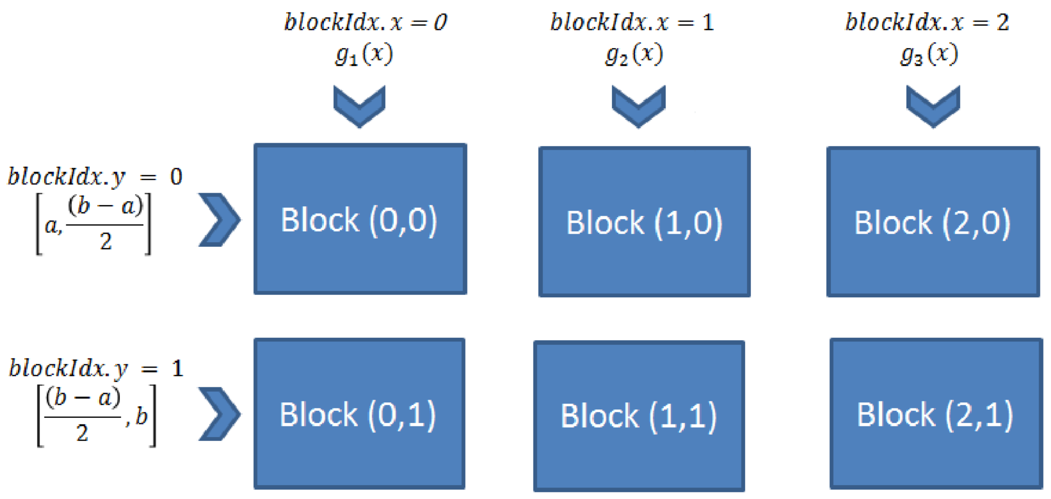


Figure 3.4: As the problem size grows more blocks are needed. This figure shows $m = 11$, $g = 3$, problem size = $2^{11} \times 3 = 6144$, Total Threads = 6144

array, Kd. Figure 3.3 shows how the area of each interval is calculated by a unique thread. A summing reduction must then be performed on the results in the device vector. The reduction using the CUDA Thrust API is used 2.3.5.The code Listing 3.1 shows the small amount of code needed for reduction with the Thrust API.

Listing 3.1: Code Required for Summing Reduction with the Thrust API

```
#include <thrust/transform_reduce.h>
#include <thrust/device_ptr.h>
#include <thrust/device_free.h>
//initialize GPU
thrust::device_ptr<Real> dev_ptr(Kd);
//execute kernel
solution = thrust::reduce(dev_ptr, dev_ptr+probSize);
```

3.2 Methodology of Test

The test methodology provides a detailed description of both systems used in this research which are listed in Section 3.2.1. The NVIDIA Tesla C2075 GPU is designed as a co-processor to be used for general purpose computing and more suited for numerical integration as compared to the NVIDIA Quadro K1000M. Any CUDA capable GPU can execute CUDA code, but the performance may vary greatly. A performance comparison of the high end Tesla and Quadro will demonstrate the disparity between GPU devices.

In order to make this comparison an appropriate timer is needed. Because GPUs are a relatively new technology the need to compare timing methods arose. NVIDIA GPUs have a startup delay that can significantly increase the execution time when it is recorded. A timing method is desired that can only time the CUDA integration calculation and not the CUDA startup delay in order to have a more appropriate comparison with the sequential quadrature method.

3.2.1 System Configurations.

Two different computer systems are provided by the AFIT QKD simulation group for this research. Each computer system has an NVIDIA CUDA capable GPU device. The specifications of each computer system is provided in Tables 3.1 and 3.2. Both systems are installed with Windows 7 (64 bit) and CUDA version 5.5. The NVIDIA Quadro K1000M is not advertised for high performance computing purposes however the Quadro is included in this research in order to show the contrast in GPU devices. The theoretical floating point operations per second (FLOPS) varies greatly between different GPU devices. The specifications for the Tesla C2075 have the theoretical FLOPS capability listed at 515 GigaFLOPS for double precision and 1030 GigaFLOPS for single precision while the Quadro K1000M FLOPS capability is not specified by NVIDIA [20].

Table 3.1: System 1 Configuration

Host Device - Intel Core i7-3610QM		GPU Device - NVIDIA Quadro K1000M	
Processor cores	8	CUDA core	1 (SM) × 192 (SP) = 192 cores
Processor Speed	2.3 GHz	GPU clock speed	0.85 GHz
Memory (RAM)	8 GB	Global memory (GDDR)	2048 MBytes
		Shared memory per block	49152 bytes
		Memory clock speed	900 MHz
		Max threads per block	1024
		Max size of each dimension of a block	[1024, 1024, 64]
		Max size of each dimension of a grid	[2,147, 483,647, 65,535, 65,535]

Table 3.2: System 2 Configuration

Host Device - Intel Xeon E5-2650		GPU Device - NVIDIA Tesla C2075	
Processor cores	8	CUDA core	14 (SM) × 32 (SP) = 448 cores
Processor Speed	2 - 2.8 GHz	GPU clock speed	1.15 GHz
Memory (RAM)	128 GB	Global memory (GDDR)	4096 MBytes
		Shared memory per block	49152 bytes
		Memory clock speed	1.57 GHz
		Max threads per block	1024
		Max size of each dimension of a block	[1024, 1024, 64]
		Max size of each dimension of a grid	[65,535, 65,535, 65,535]

The Tesla GPU is considered by NVIDIA to be a *co-processor* and has the advantage of being solely dedicated to computational work [20]. Work stations with an all-in-one GPU device may have limited performance because the device is also being used to render graphics.

The Quadro has a timeout error after two seconds of execution due to a timeout detection and recovery feature of the Windows operating system. The timeout detection and recovery “detects response problems from a graphics card, and recovers to a functional desktop by resetting the card. If the operating system does not receive a response from a graphics card within a certain amount of time (default is 2 seconds), the operating system resets the graphics card” [21]. When the graphics card is reset any data on the GPU device is lost. The timeout detection and recovery feature can be disable or changed to allow more time before a timeout occurs [21]

3.2.2 Performance Metrics.

The primary metric is execution time and relative speedup of the integrations being performed. Execution time is measured by timers (discussed further in Section 3.2.3) in milliseconds. Speedup, S, is the ratio of the serial run-time of the best sequential integration calculation, T_s, to the run-time of the parallel integration calculation, T_p (Equation 3.3) [22].

$$S = \frac{T_s}{T_p} \tag{3.3}$$

The solution from the CUDA quadrature method will also need to converge to within ϵ less than 10^{-10}. Equation 3.4 provides an equation for evaluating convergence. The CUDA integration method is said to converge if the absolute of the difference in solutions between refinement steps reaches a value of less than ϵ.

$$|x_{k+1} - x_k| < \epsilon \tag{3.4}$$

3.2.3 Choosing a Timing Method.

A timing method is desired that can accurately and consistently time calculations within half a millisecond. Four different timing methods are compared to determine which is the most suitable for evaluating the CUDA integration technique, as well as sequential integration. The four timing methods are the CUDA event timer (CET) which utilizes CUDA events and three timers (timer 1, 2 and 3) that utilize the Windows system API high resolution counter, `QueryPerformanceCounter()` and system frequency, `QueryPerformanceFrequency()`. Statistical comparisons are performed on each timer and the results are provided in Appendix A. The results from the test reveal there is no statistical advantage to using the CET, however, there are other advantages to using the CET that will be discussed in the remainder of this section.

3.2.3.1 CUDA Event Timer.

?? The CET is shown, in Listing 3.2. The function `cudaEventCreate(cudaEvent_t)` creates a CUDA event object from the CUDA event type, `cudaEvent_t`. An event is recorded using `cudaEventRecord(cudaEvent_t,cudaStream_t)` after all proceeding CUDA operations have completed. The stream is set to zero as the default stream. The `cudaEventRecord(cudaEvent_t, cudaStream_t)` is an asynchronous function and returns immediately regardless if the cuda event has been recorded. In some scenarios it is necessary to use `cudaEventSynchronize(cudaEvent_t)` to block until the event has been recorded. An event is recorded before and after the calculation. The function `cudaEventElapsedTime(*float,cudaEvent_t,cudaEvent_t)` computes the time that has elapsed between the two events to about half a millisecond [23]. It is important to note that the CUDA kernel is asynchronous and `cudaThreadSynchronize()` must be called after the kernel in order to block the host until all CUDA threads have completed. Otherwise, the kernel returns immediately and the timing is inaccurate.

Listing 3.2: CUDA Event Timer Code

```
float time;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
cudaEventSynchronize(start); //OPTIONAL

//Calculate Something
```

```

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

3.2.3.2 CUDA Event Synchronize.

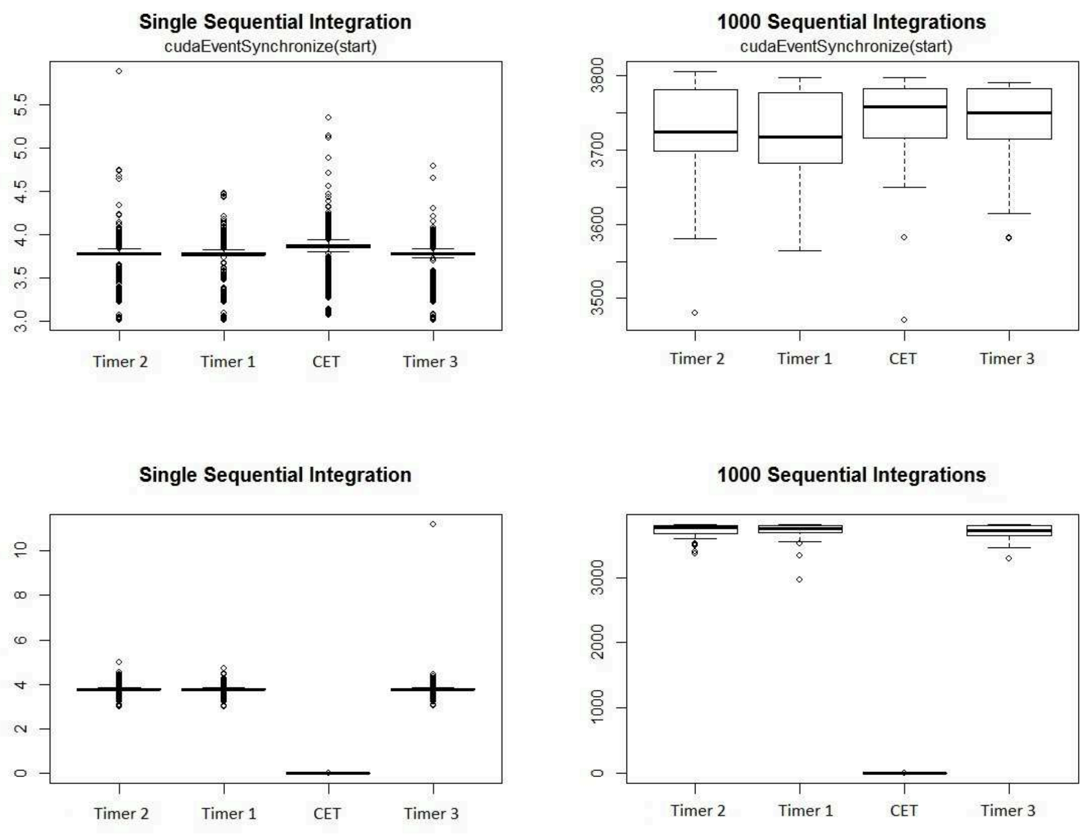


Figure 3.5: Sequential Timer Experiment with and without Synchronizing CUDA Events

It is observed that unless the a synchronize call is performed after the *start* event, `cudaEventSynchronize(start)`, the CET produces poor results. However, when the synchronize function is performed, the CET results more closely represent the results produced from the other timers. Figure 3.5 shows the effects of not using `cudaEventSynchronize(start)`.

3.2.3.3 GPU Startup Delay.

A startup delay is consistently observed at the first integration regardless of the timer used. Figure 3.6 shows the first timer consistently having a outlier close to 150 to 200 ms. Startup delays of up to 800 milliseconds are observed. The CET did not have any extreme outliers. The startup delay is caused by the CET makes calls to the GPU which begins the GPU startup process. Using the CET eliminates recording the startup delay as part of the calculation time. The NVCC documentation states that startup delay can be reduced by generating code for multiple GPU device types which can be set in the NVCC compiler options [24]. The focus of this study is to evaluate the time required for integration, therefore, the startup delay time is not included in the time per integration.

Because the CET did not include startup delay and because its performance was no worse than the other three timers it is selected as the primary timer for evaluating the performance of the GPUs and CPUs. The remaining tests and experiments only utilize the CUDA event timer.

3.3 Methodology Summary

This chapter described the design and test methodology for this research. The primary goal is to determine if CUDA programming can reduce execution time of integration of ID300 optical pulses representations in the QKD simulation. A sequential quadrature algorithm is developed from [15]. The CUDA quadrature algorithm is designed with scalability in mine. The problem size as well as the number of Gaussian pulses can vary.

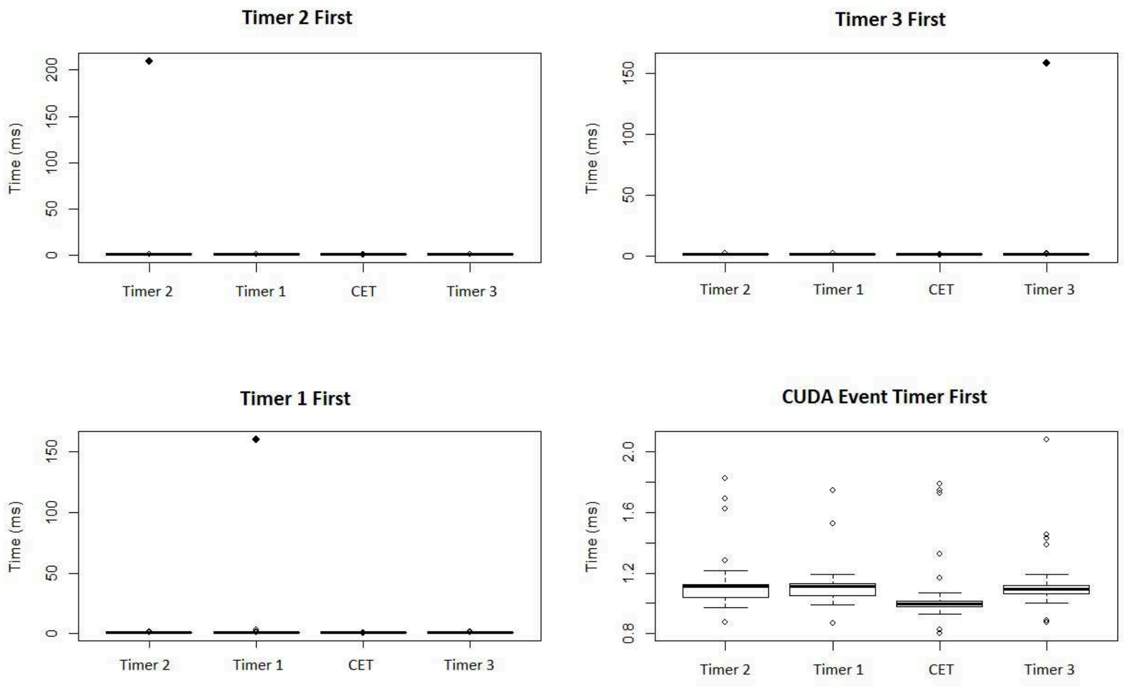


Figure 3.6: CUDA Timer Experiment Order of Timer

The test methodology provides a investigation into why the CUDA event timer is selected as the optimal timer for this research. In addition a definition of the metrics used to evaluate CUDA quadrature, speedup and convergence, are provided.

IV. Results and Analysis

THE results obtained from integrating the ID300 pulse with CUDA and sequential quadrature methods are presented in this chapter. The execution time for both methods is recorded using the CET (Section 3.2.3). The sequential and CUDA execution times are used to calculate speedup. In addition, the solutions obtained per problem size are provided with figures showing the convergence for each device used.

4.1 CUDA Integration Calculation

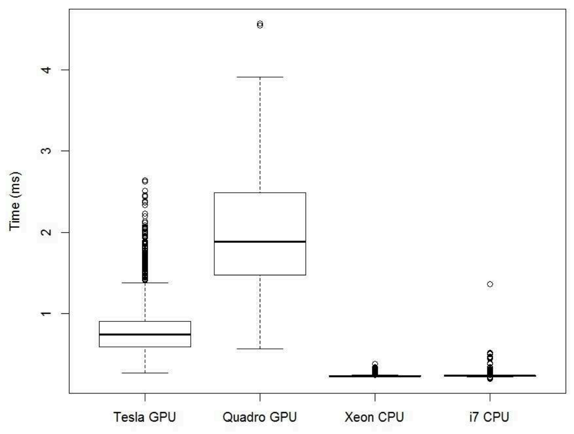
The CUDA kernel performs all the point calculations to determine the solution, however, before the kernel can be executed memory space on the GPU device must be allocated and the grid dimensions must be determined. All of the kernel setup operations are handled by the constructor for the `cudaIntegral` class, `cudaIntegral(int numGaus, Real A, Real B, int m)`.

Table 4.1: One-sided t-test, Kernel Mean < Sequential Mean

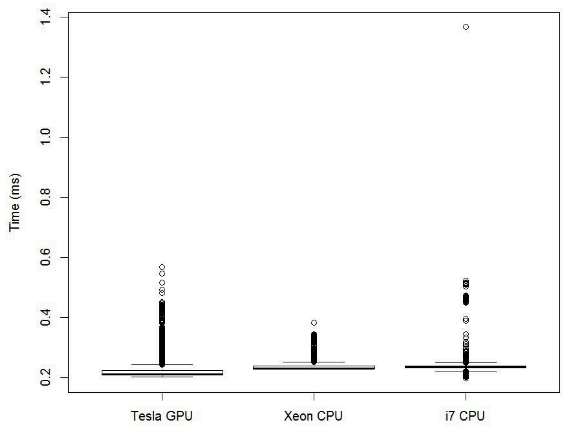
n = 1, r = 10,000

	95% Confidence Interval	p-value	Kernel Mean (ms)	Seq. Mean (ms)
Tesla vs. Xeon	$(-\infty, -0.01782292)$	$< 2.2e^{-16}$	0.2224929	0.2409594
Tesla vs. i7	$(-\infty, -0.02018621)$	$< 2.2e^{-16}$	0.2224929	0.2435304
Tesla vs. (Xeon & i7)	$(-\infty, -0.01911166)$	$< 2.2e^{-16}$	0.2224929	0.2422449

Once the cudaIntegral object has been initialized with the constructor the kernel can be executed as many times as necessary. Figure 4.1 shows a comparison between the GPU devices and CPUs for calculating a single integral using 1024 points ($m = 2^{10}$). The kernel execution time of the Tesla is comparable to the sequential time of the Xeon and i7 CPU. The one-sided t-test (Table 4.1 indicates that the Tesla kernel is at least seventeen microseconds faster than the Xeon CPU and that the Tesla kernel is on average 20 microseconds faster than the Xeon and i7 CPU when considering the CPUs as a single group. As more and more integrals are performed the time saved begins to add up.



(a) With Constructor



(b) Kernel Only

Figure 4.1: CUDA Execution Time, n = 1, r = 10,000, timed with the CUDA event timer (CET)

4.2 Scalability of CUDA Integration Calculation

Several tests are performed to evaluate the speedup of GPU as the number of points(and intervals) per integration is increased. In addition the number of Gaussian functions per pulse is increased which increases the problem size further. Results using single and double precision are compared to determine if there is a speed advantage to using either double or single precision. Lastly the solution from double and single precision are evaluated to determine which level of precision is required.

4.2.1 Points Calculated per Pulse.

The number of points calculations per integration is varied for sequential and CUDA integration from $2^m = 2^{10}$ to 2^{25} using the standard pulse. Each integral is repeated ten times (n = 10, r = 10). Figure 4.2 shows that CUDA integration with the Tesla GPU outperforms all other devices. As the number of points calculated per integration increases from 2^{10} to 2^{25} the Tesla Execution time increases at a much lower rate as compared to the other devices. The i7 processor has the worst ability to scale out of the group of hardware evaluated. The same data is shown with a logarithmic scale on the y-axis in Figure 4.3. Table 4.2 shows the speedup of the Tesla compared to the Xeon processor. CUDA has the capability of using double and single precision Figure 4.4 shows the difference in using the two data types. As the problem size becomes very large the single precision (float) execution begins to have a reduced execution as compared to double precision.

Table 4.2: Mean Speedup of Tesla compared to Xeon

for n = 10, r = 10

m	10	12	14	16	18	20	22	24
Speedup	1.012578	2.338682	5.684155	10.74699	17.91386	20.85142	21.61606	22.06557

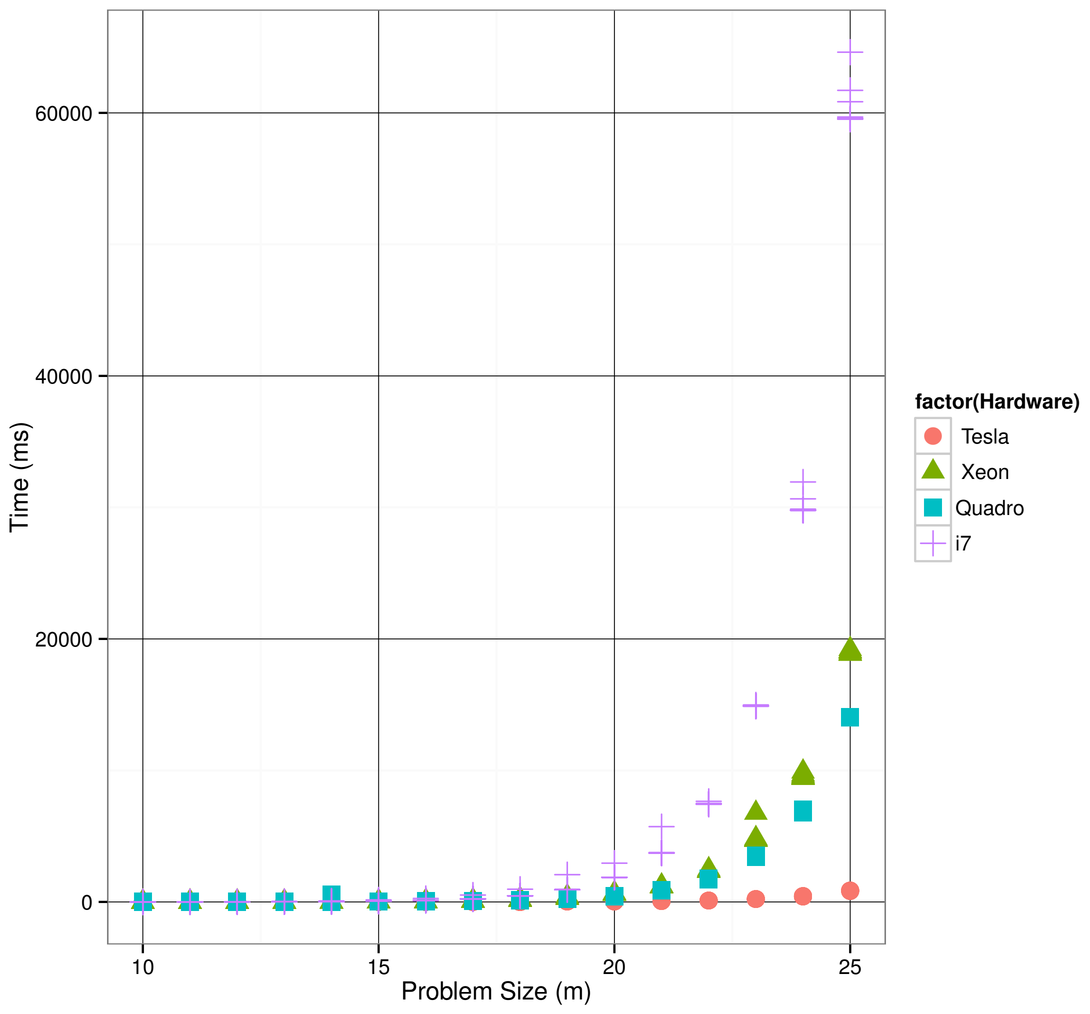


Figure 4.2: System Comparison, Execution Time in (ms) vs. Problem Size (2^m), n = 10

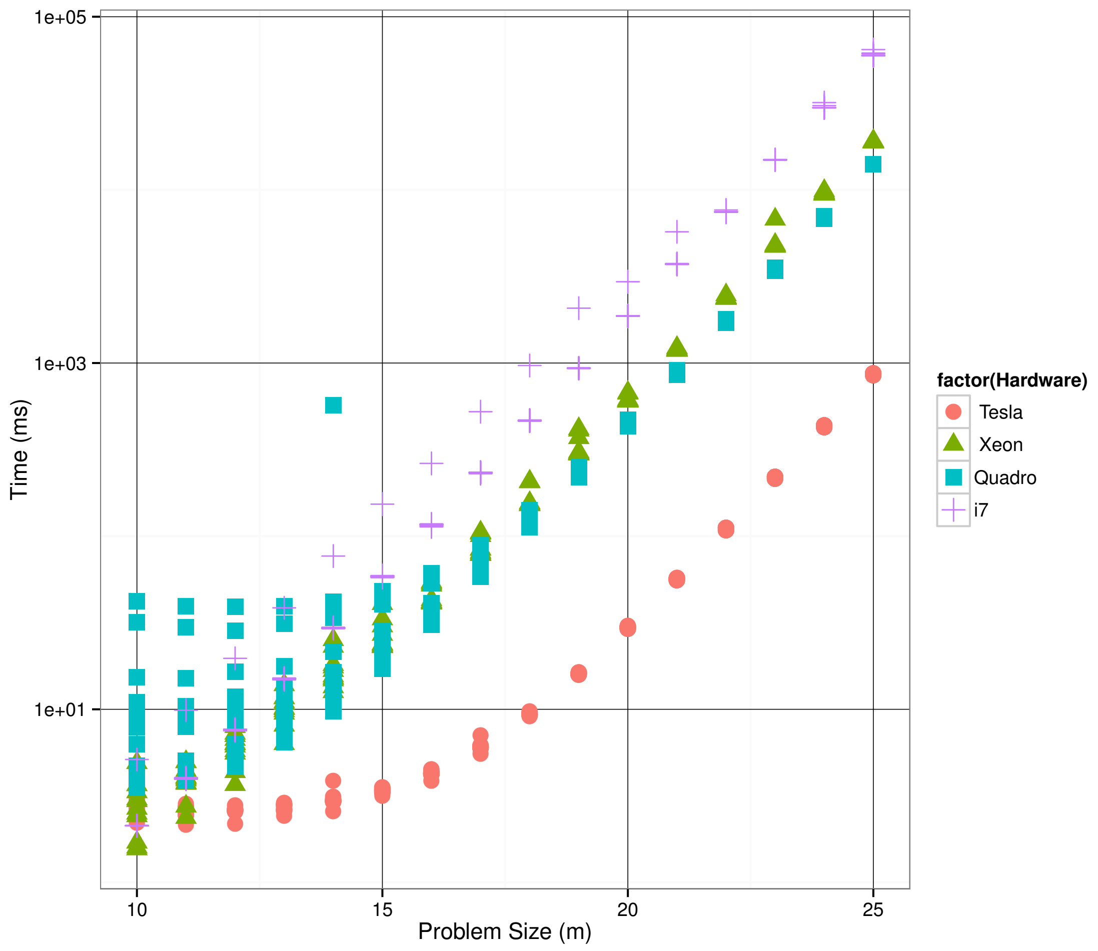


Figure 4.3: System Comparison, Execution Time (ms) vs. Problem Size (2^m), n = 10 y-axis (log scale)

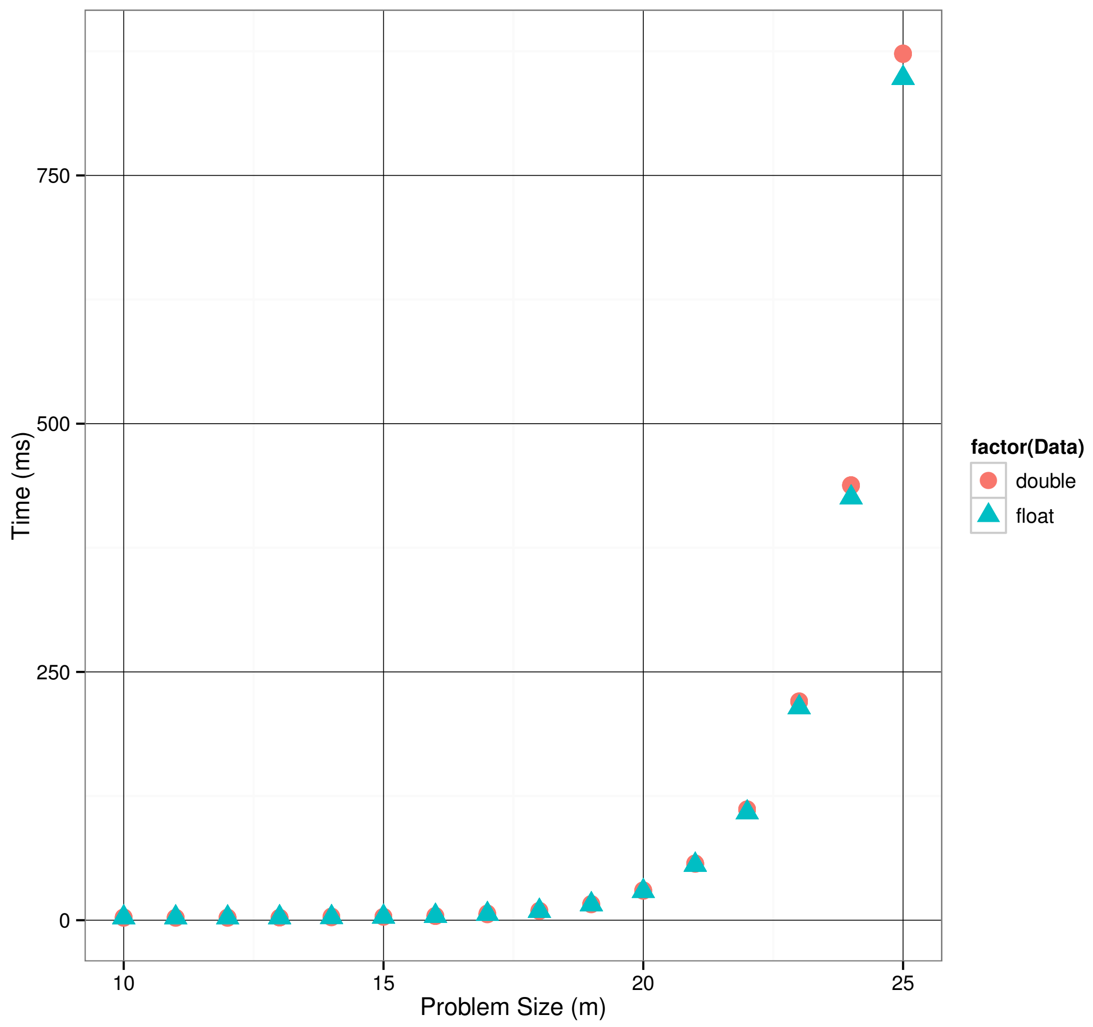


Figure 4.4: Precision, Execution Time (ms) vs. Problem Size (2^m) on Tesla GPU, n = 10

4.2.2 Increased Gaussians per Pulse.

The standard pulse, with three Gaussians, is currently being used in the QKD simulation to model a pulse. However, future QKD simulations may require a pulse model with more than three Gaussians. The number of Gaussians within a pulse increased from $g = 3$ to $g = 15$ and the execution time is recorded for $n = 10$ integrals with 10 replications ($r = 10$). Figure 4.5 shows that the the Tesla GPU greatly outperforms the i7 and Xeon processors as the number of Gaussians is increased. Table 4.3 provides the relative speedup of the Tesla compared to the i7 processor.

Table 4.3: Mean Seedup of Tesla compared to i7

for n = 10, r = 10

m	10	12	14	16	18	20
Speedup g = 3	0.7554481	3.052879	9.360489	28.45976	52.27691	63.99778
Speedup g = 6	1.920524	6.434797	18.81976	46.74105	69.98963	66.19936
Speedup g = 9	2.7045	8.725577	24.10778	45.0548	58.23774	63.12008
Speedup g = 12	3.392989	11.11147	32.08359	51.7936	62.84155	64.76208
Speedup g = 15	3.793517	12.69264	30.59965	51.10351	60.35495	63.2263

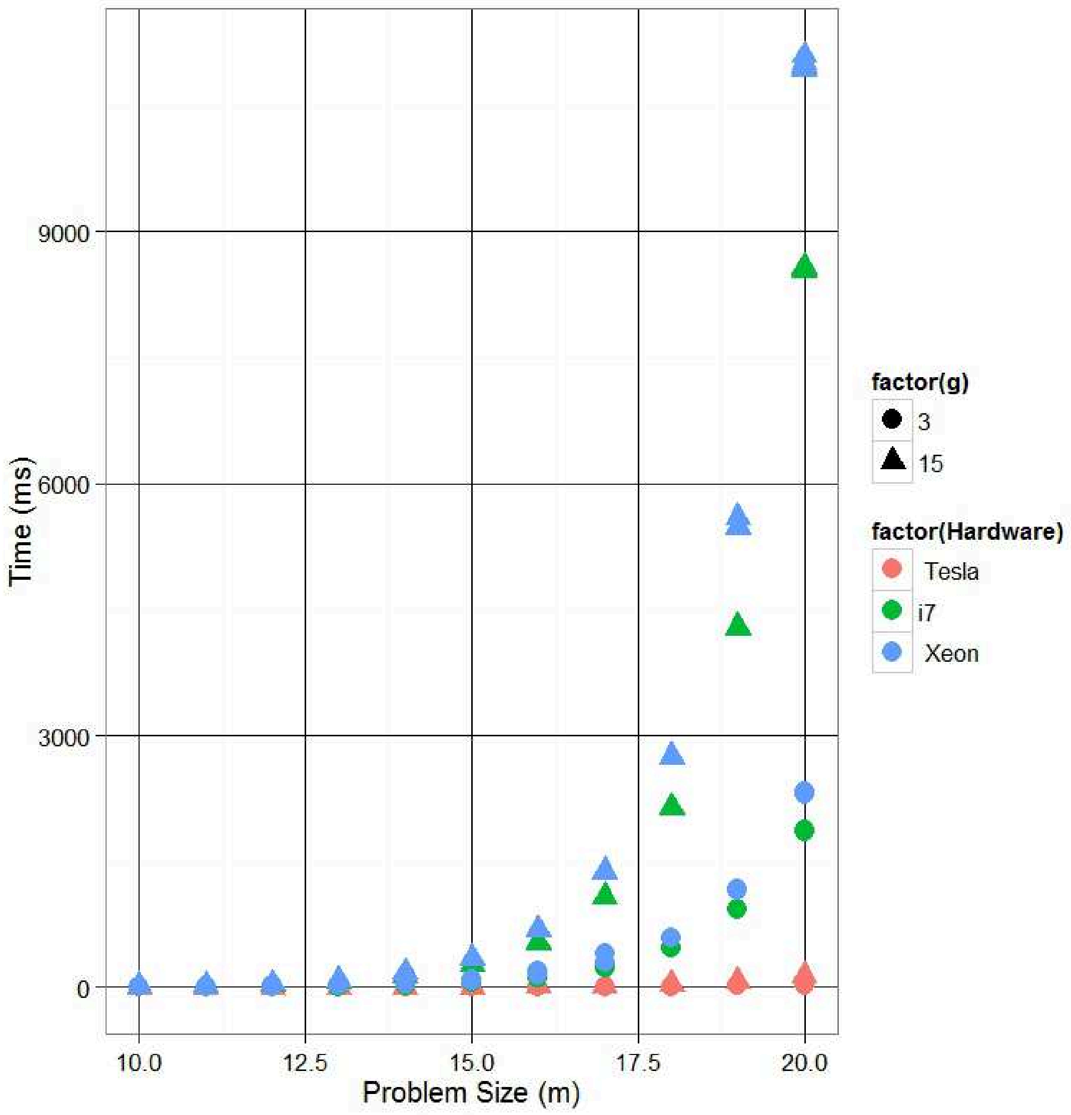


Figure 4.5: Number of Gaussians per Pulse Increased from g = 3 and g = 15 on Tesla, i7 and Xeon, n = 10

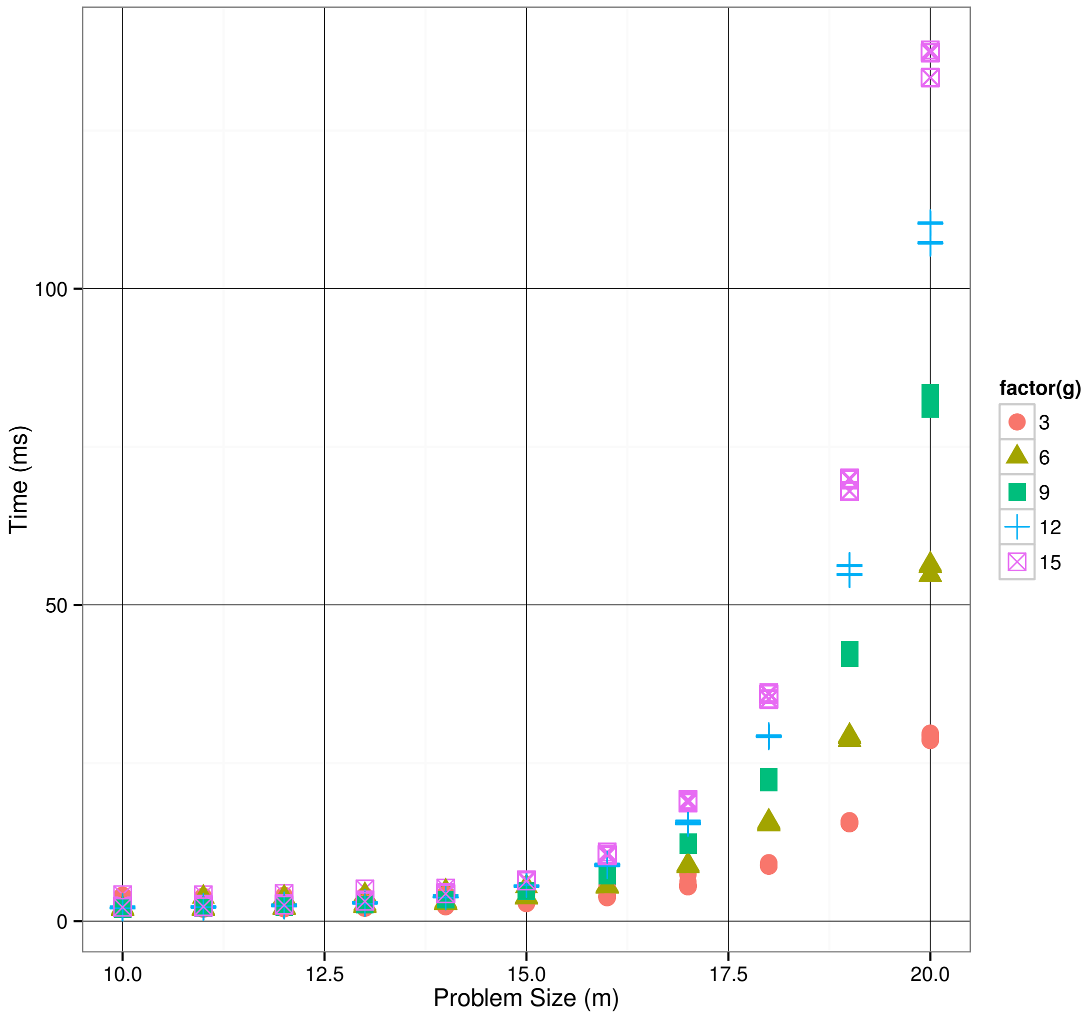


Figure 4.6: Number of Gaussians per Pulse Increased g = 3 to g = 15 on Tesla GPU, n = 10

4.3 Accuracy and Convergence

There is no analytically solution to the integration of Gaussian shapes. Thus the true accuracy of the CUDA quadrature method cannot be known. However evaluating the convergence provides some assurance that the numerical solution is trustworthy. The nature of Guassian pulses (tapering off at the ends) indicate that the numerical integration should converge to a solution. The method for determining convergence is defined in Section

3.2.2. Figures 4.7 - 4.9 show the convergence of the i7 CPU and Tesla GPU. When using single precision, floating point numbers, the CUDA quadrature method does not converge to withing 10^{-10}.

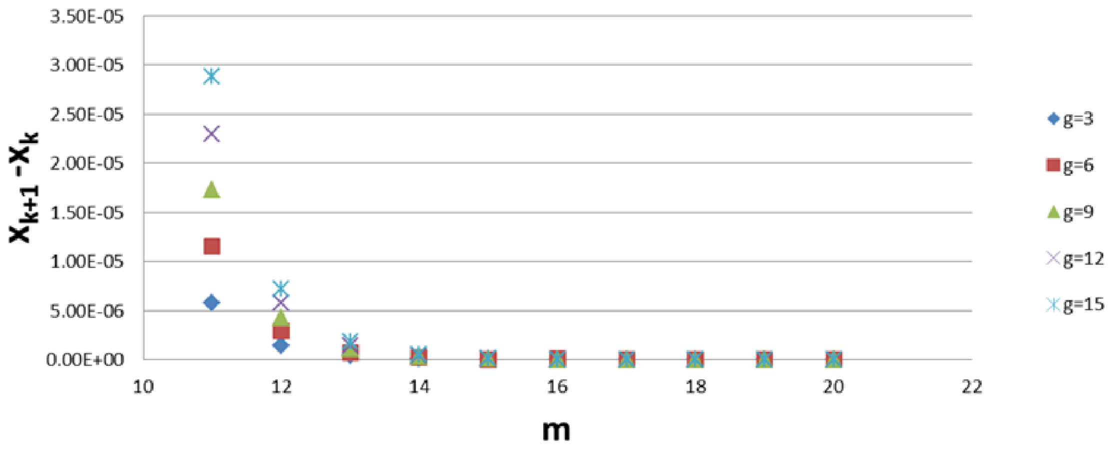


Figure 4.7: The convergence of the sequential algorithm is shown using the results from i7 CPU using double precision. As the problem size grows the difference in the solution approaches zero.

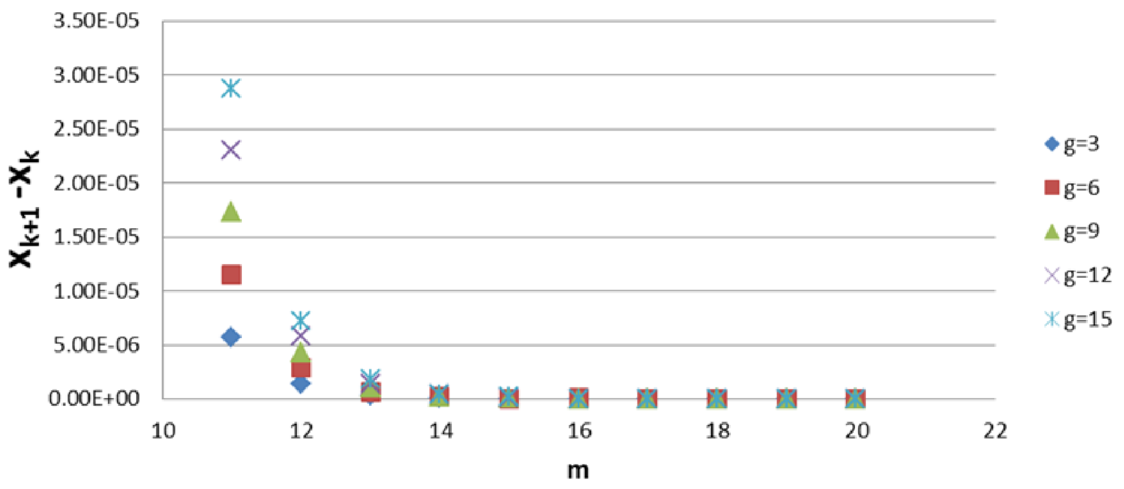


Figure 4.8: The CUDA algorithm converges to zero. The results shown are from Tesla GPU using double precision.

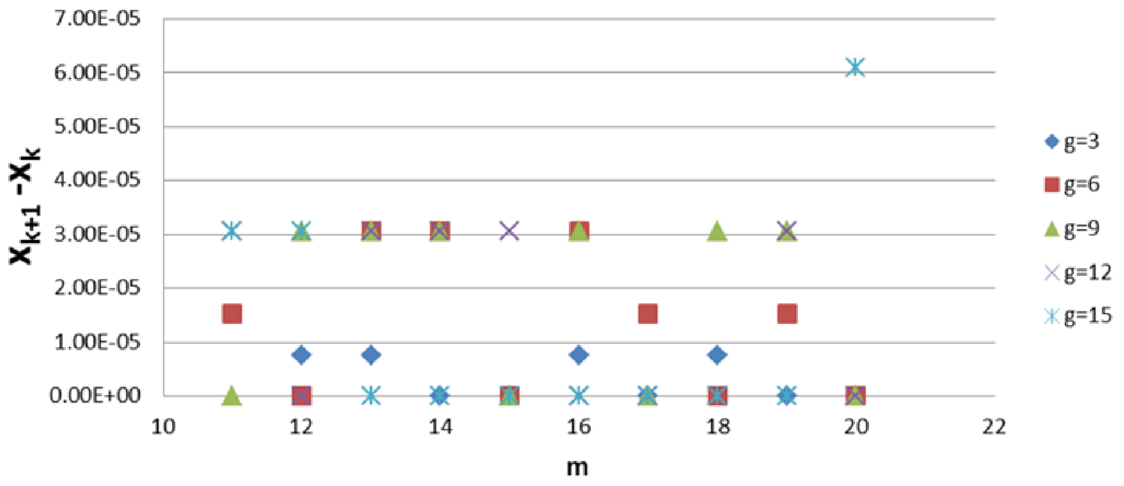


Figure 4.9: When using single precision the CUDA algorithm does not converges to zero. The results shown are from Tesla GPU using single precision.

4.4 Results and Analysis Summary

The results obtained by varying the problem size and the number of Gaussians per optical pulse is presented in table and graphical form. Comparing the execution time of the sequential quadrature to the CUDA quadrature a speedup for each problem size is determined. The for problem sizes about 2^{10} the Tesla GPU using CUDA quadrature has a faster execution time as compared to the i7 and Xeon CPU. As the number of Guassians per optical pulse is increased the problem size is increased and the speedup achieved by the Tesla compared to the i7 and Xeon increases. Using single precision with CUDA quadrature only achieves a speedup for very large problems sizes. The small about of speedup is not significant because the CUDA quadrature using single precision does not converge to withing $\epsilon = 10^{-10}$.

V. Conclusion

A simple approach to performing numerical integration using CUDA capable GPU devices is presented for the ID300 optical pulse. The results show that GPUs can achieve better execution times than sequential algorithms when performing integration using the trapezoidal method. It was found that integration of the ID300 pulse with $2^{10} = 1024$ intervals and three Gaussian functions produced a very small speedup. On average the Tesla kernel outperforms the Xeon and i7 processor by at least 19 microseconds for a single integration calculation (Table 4.1). However, when the time for constructing the CUDA integrator is included in the execution time for a single integral the sequential program has a faster execution time (Figure 4.1).

The problem size is scaled to 2^{25} intervals between a and b. As the problem size increases, the performance of the CUDA integration method improves. As seen in Table 4.2 a speedup of up to 22x is achieved for 10 integrations per replication. The rate of growth of the Tesla GPU is much less than all other devices in this study. As the number of Gaussian functions is increased in addition to the problem size the speedup observed is much higher, exceeding 60 times. The Tesla device can easily handle thousands to millions of threads in a single execution, demonstrating excellent scalability.

When comparing the execution time of double single precision there is little difference. The results show that the single precision solution is significantly inconsistent with the double precision solution. There is no substantial benefit to using single precision for CUDA numerical integration. Because single precision does not have a significantly improve the execution time double precision is recommended.

Several objectives listed in Chapter 1 have been met. A CUDA integration software using C++ was developed than can easily be integrated into the existing QKD simulation because it is contained in a C++ class. A speedup greater than one was achieved for problem sizes greater than 2^{10}. With the addition of more Gaussian shapes within a pulse the speedup ranges from 1.9x to greater than 60x. However, the CUDA quadrature method does not have the capability of replacing the sequential quadrature because even though very high speedups are achieved they offer no benefit because the solution does not improve at these high levels of refinement. More research is required to determine the most appropriate method for integrating the ID300 pulse as part of the QKD simulation.

5.1 Future Research

Future research in this area should focus on implementing alternative quadrature methods for optical pulses with CUDA. Alternative quadrature methods may have a faster convergence for the optical pulse shape, which will require fewer calculations. A GPU adaptive quadrature method is attempted by [17] (Section 2.4.2) who reported the execution time was doubled. Thus, adaptive quadrature is not practical for GPU implementation. However, other Newton-Cotes methods, such as Simpson's rule, could easily be adaptive into the CUDA integration software developed. In [19] Simpson's rule was implemented using GPU for integrating financial pricing function and reported Simpson's rule has a faster convergence. Implementations of more advanced integration techniques such as non-adaptive Gaussian Quadrature are also feasible. However, it is less likely there will any speedup because Gaussian Quadrature does not require as many intervals as the Trapezoid Rule.

Other areas of research include investigating integration methods that take advantage of the Gaussian model of the ID300 pulse. One such approach is to use the Error function to solve the integral of each Guassian. Preliminary results using the error function show the integration time orders of magnitude faster than the sequential and CUDA quadrature presented in this research.

5.2 Lessons Learned in GPU Programming

GPU and sequential programming are very different. There are many challenges associated with GPU programming that do not exist in sequential programming. The following list is a summary of the lessons learned programming with CUDA on Windows.

- The problem should have data parallelism and the problem size should be very large.
- Avoid thread branching within a kernel by avoiding if-then-else statements within a kernel.
- Copying into and out of the GPU device is a very slow operation. Avoid costly copying by reducing the solution within a block or thread as much as possible using shared memory.
- Investigate using CUDA libraries. Many of the available libraries simplify CUDA programming to the point where the user does not need an extensive knowledge of CUDA or GPUs.
- Use a dedicated GPU for computational work. If a dedicated GPU is not used the GPU will timeout after a CUDA execution exceeds two seconds.

Appendix: Timer Comparison Experiment

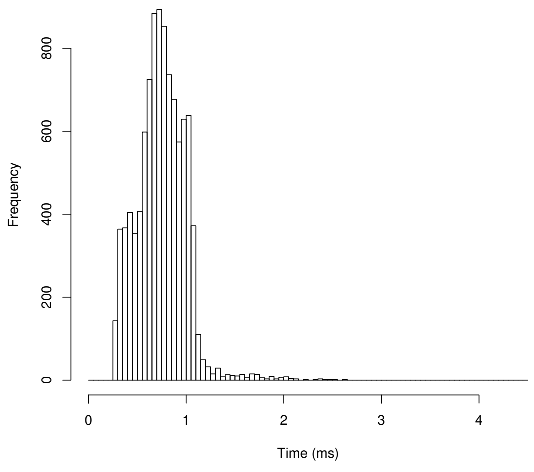
Tests are performed to determine if there is a significant difference between the timers described in Section 3.2.3.

A.1 Timer Test for a Single Integration

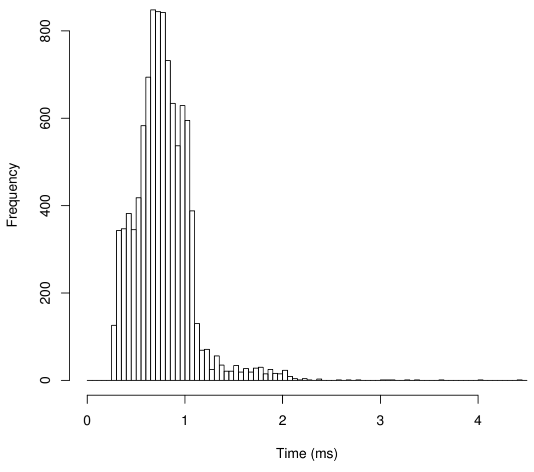
The test records the time for one integration (n=1) of a pulse to complete and is replicated 10,000 times (r = 10,000). The test is conducted on the system 1 and 2 GPUs (Tesla and Quadro) for integration using CUDA and on the system 1 and 2 CPUs (Xeon and i7) for sequential integration. The results from the Tesla GPU running the CUDA algorithm and the Xeon CPU running the sequential algorithm are shown in Figures A.1 and A.2.

The data shown in Figures A.1 and A.2 is not normally distributed. Using the Central Limit Theorem, a sampling distribution of the averages is determined by sampling the distributions in Figures A.1 and A.2 100 times. The 100 samples are averaged and the sampling process is repeated 1000 times to form the normal distributions in Figures A.3 and A.4.

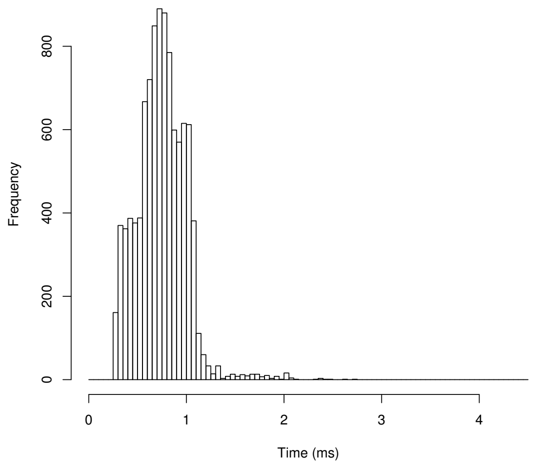
A two-sided t-test was conducted on the sampled distribution of averages in order to determine if there is a statistical difference between the CUDA event Timer and the three additional timers for timing a single integration. The results from the t-test are shown in Table A.1. The low p-values ($p - value < 0.05$) for each test suggests that there is a significant difference between the CET and Timers 1, 2 and 3. The mean difference between the CET and one of the other three timers lies between 30 and 0.8 microseconds when timing calculations on the Tesla GPU. Timing calculations on the Xeon CPU the mean difference between timers has a smaller range, 0.09 and 9 microseconds. In addition, Figure A.1 shows the boxplots of the data in Figures A.3 and A.4 which allow for a visual comparison of sampled means.



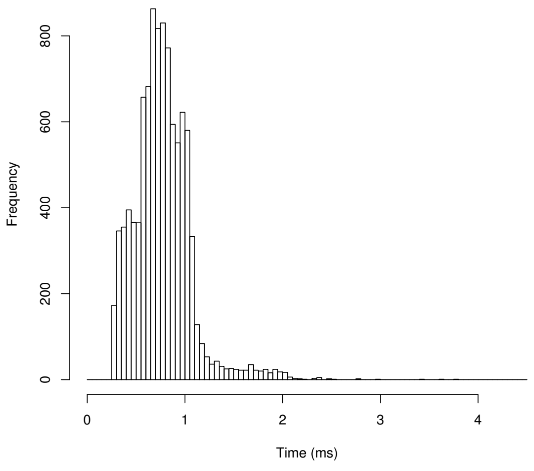
(a) Cuda Event Timer



(b) Timer 1

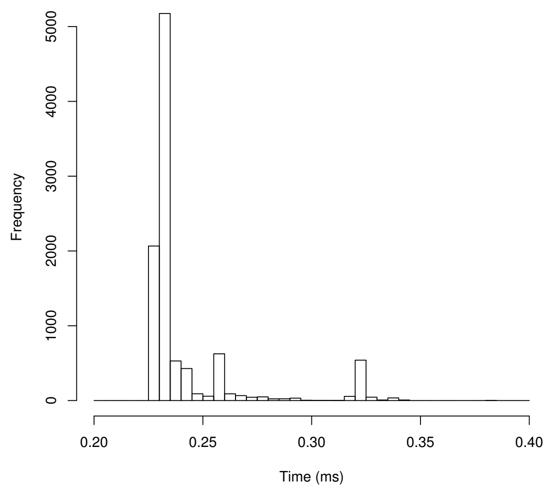


(c) Timer 2

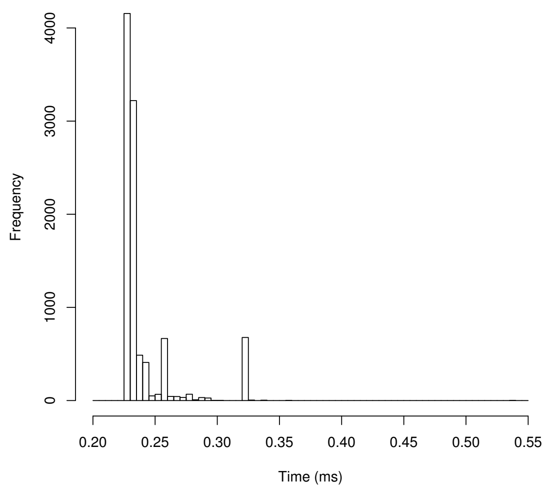


(d) Timer 3

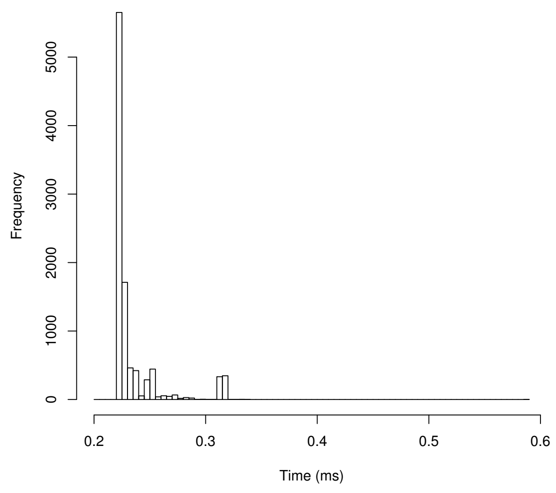
Figure A.1: Timer Test on Tesla GPU, a single integration is replicated 10,000 times (n = 1, r = 10,000)



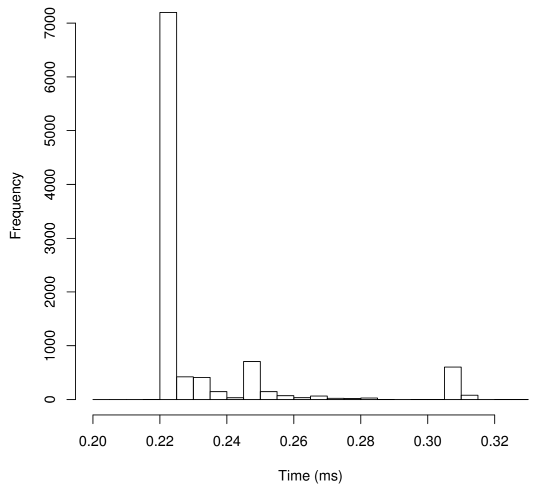
(a) Cuda Event Timer



(b) Timer 1

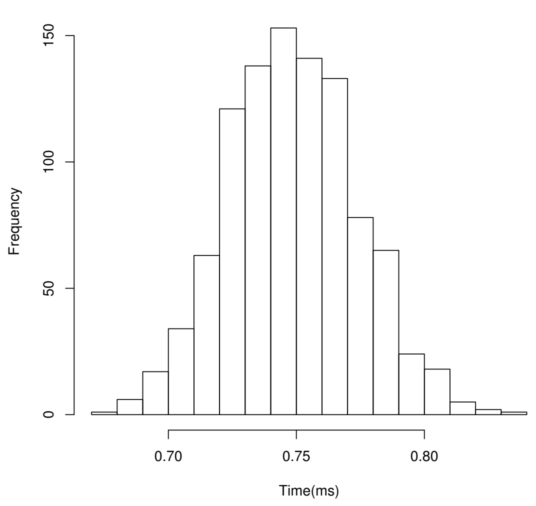


(c) Timer 2

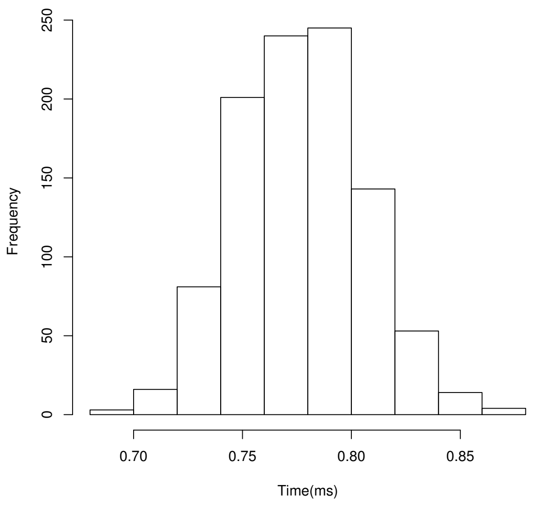


(d) Timer 3

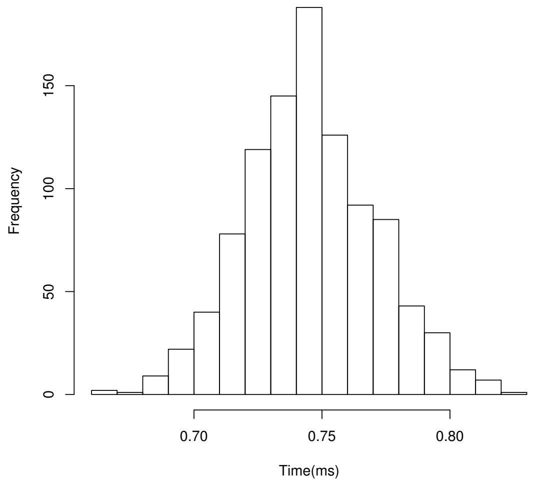
Figure A.2: Timer Test on Xeon CPU, a single integration is replicated 10,000 times (n = 1, r = 10,000)



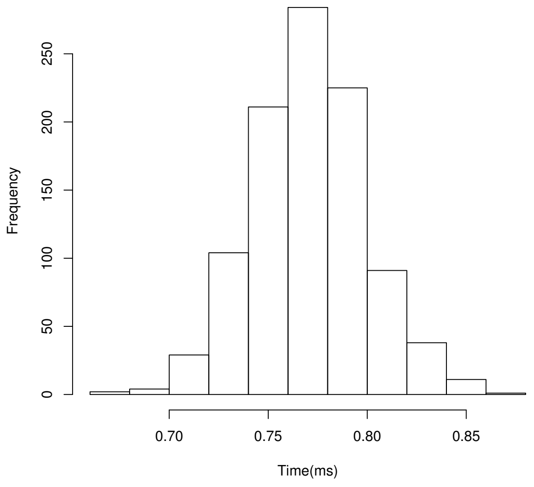
(a) Cuda Event Timer Averages



(b) Timer 1 Averages

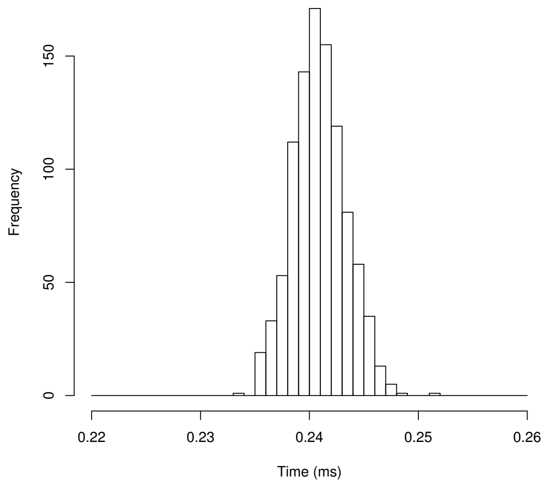


(c) Timer 2 Averages

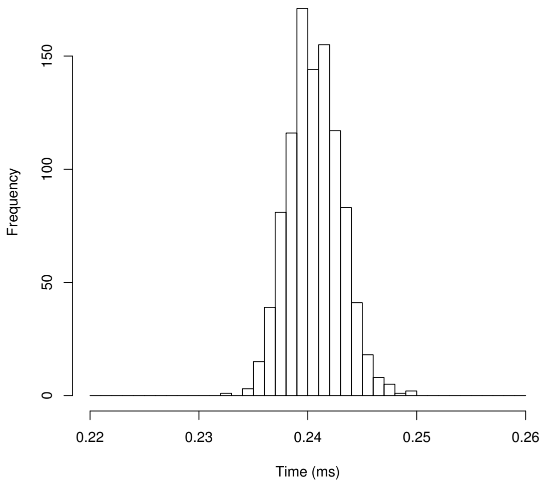


(d) Timer 3 Averages

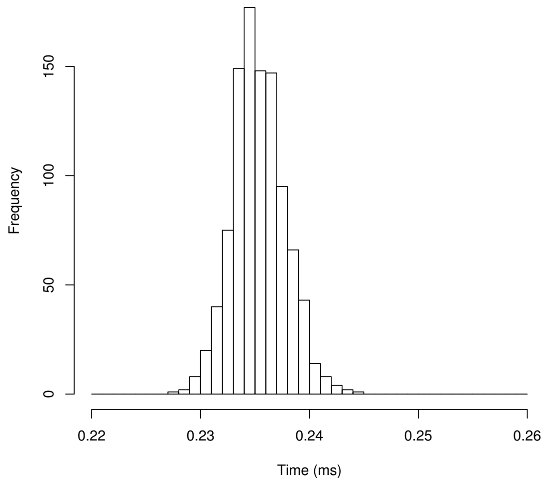
Figure A.3: The sampling distribution of the averges on the Tesla GPU form a normal distribution



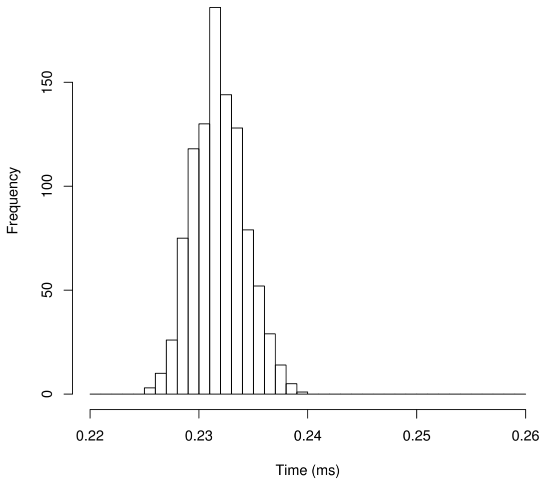
(a) Cuda Event Timer Averages



(b) Timer 1 Averages



(c) Timer 2 Averages

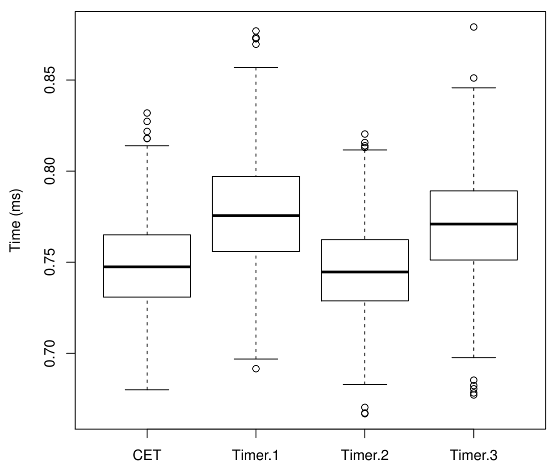


(d) Timer 3 Averages

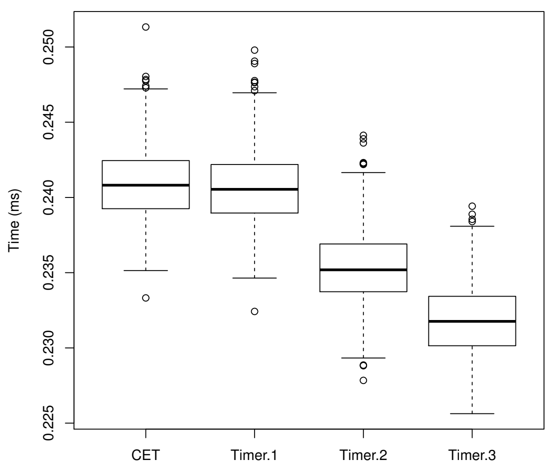
Figure A.4: The sampling distribution of the averges on the Xeon GPU form a normal distribution

Table A.1: Two-sided t-test for CUDA Event Timer compared to Timers 1, 2, and 3

Device	Null Hypothesis	95% Confidence Interval	p-value	Mean CET	Mean Other
Tesla	$\mu_{CET} = \mu_{T1}$	(0.03054815, 0.02575845)	$< 2.2e^{-16}$	0.7486111	0.7767644
Tesla	$\mu_{CET} = \mu_{T2}$	($8.368394e^{-4}$, $5.2587512e^{-3}$)	0.006921	0.7486111	0.7455633
Tesla	$\mu_{CET} = \mu_{T3}$	(0.02426579, 0.01954475)	$< 2.2e^{-16}$	0.7486111	0.7705163
Xeon	$\mu_{CET} = \mu_{T1}$	($9.472832e^{-5}, 5.169560e^{-4}$)	0.004541	0.2409090	0.2406032
Xeon	$\mu_{CET} = \mu_{T2}$	($5.347151e^{-3}, 5.770945e^{-3}$)	$< 2.2e^{-16}$	0.240909	0.235350
Xeon	$\mu_{CET} = \mu_{T2}$	($8.832244e^{-3}, 9.251133e^{-3}$)	$< 2.2e^{-16}$	0.2409090	0.2318674



(a) Tesla GPU sampled distribution of averages



(b) Xeon CPU sampled distribution of averages

Figure A.5: Boxplot data for sampled distribution of averages

A.2 Timer Comparison Summary

Even though the CET is significantly different from other timers the variation between Timers 1, 2, and 3 implies that they are not equivalent. A t-test on Timers 1, 2, and 3 data from the Tesla GPU reveals that Timers 1, 2, and 3 are not identical timers (*p-value* ¡ 0.05,

Table A.2: Two-sided t-test Timers 1, 2, and 3 on Tesla

a	b	95% Confidence Interval	p-value	μ_a	μ_b
Time 1	Timer 2 and 3	(0.01648331, 0.02096581)	$< 2.2e^{-16}$	0.7767644	0.7580398
Timer 2	Timer 1 and 3	(-0.03010360, -0.02605056)	$< 2.2e^{-16}$	0.7455633	0.7736404
Timer 3	Timer 1 and 2	(0.007101049, 0.011603990)	$6.258e^{-16}$	0.7705163	0.7611638

the true difference in means is not zero). There is some variation even among timers all using Windows system API. Because the CET timer variation from the other timers is very similar to the variation among Timers 1, 2, and 3. It can be said that statistically there is no advantage to using any one of the three timers.

Bibliography

[1] M. R. Grimaila, J. Morris, and D. Hodson, "Quantum key distribution: A revolutionary security technology," *The Information System Security Association Journal*, vol. 10, no. 6, pp. 20–27, 2012.

[2] A. Varga *et al.*, "The omnet++ discrete event simulation system," in *Proceedings of the European Simulation Multiconference (ESM2001)*, vol. 9, p. 185, sn, 2001.

[3] NVIDIA Corporation, *CUDA Programming Guide*, 5.0 ed., 2013. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[4] S. Singh, *The code book: The secret history of codes and code-breaking*. London: FourthEstate, 17 ed., 1999.

[5] B. Schneier, *Applied cryptography: Protocols, algorithms, and source code in C*. New York: John Wiley & Sons, Inc, 18 ed., 1995.

[6] S. and W. K. Wootters, *Protecting information: From classical error correction to quantum cryptography*. New York: Cambridge University Press, 1 ed., 2006.

[7] C. H. Bennett and G. Brassard, "Quantum cryptography: Public key distribution and coin tossing," in *International Conference on Computers, Systems & Signal Processing*, 1984.

[8] "Cuda community showcase." http://www.nvidia.com/object/cuda_showcase_html.html. NVIDIA Corporation, Accessed: 2013-09-05.

[9] D. Kirk and W.-m. Hwu, *Programming MassivelyParallel Processors: a hands-on approach*. Burlington, MA: Morgan Kaufmann Publishers, 2010.

[10] A. Goshtasby and O. William, "Curve fitting by a sum of gaussians," *CVGIP: Graphical Models and Image Processing*, vol. 56, no. 4, pp. 281–288, 1994.

[11] L. Lydersen, N. Jain, C. Wittmann, Ø. Marøy, J. Skaar, C. Marquardt, V. Makarov, and G. Leuchs, "Superlinear threshold detectors in quantum cryptography," *Physical Review A*, vol. 84, no. 3, p. 032320, 2011.

[12] "Gpu-accelerated libraries." https://developer.nvidia.com/gpu-accelerated-libraries. NVIDIA Corporation, Accessed: 2014-03-05.

[13] M. Harris, *Optimizing Parallel Reduction in CUDA*. NVIDIA Developer Technology, June 2013. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[14] "Nvidia developer zone." https://developer.nvidia.com/thrust. NVIDIA Corporation, Accessed: 2013-02-14.

[15] W. H. Press, *Numerical recipes: The art of scientific computing*. Cambridge, UK; New York: Cambridge University Press, 3rd ed., 2007.

[16] J. Place and J. Stach, "Efficient numerical integration using gaussian quadrature," *Simulation*, vol. 73, no. 4, pp. 232–237, 1999.

[17] B. Nelson, R. Kirby, and R. Haimes, "Gpu-based volume visualization from high-order finite element fields," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 1, p. 70, 2011.

[18] K. Arumugam, A. Godunov, D. Ranjan, B. Terzic, and M. Zubair, "An efficient deterministic parallel algorithm for adaptive multidimensional numerical integration on gpus," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, pp. 486–491, IEEE, 2013.

[19] A. H. Tse, D. Thomas, and W. Luk, "Design exploration of quadrature methods in option pricing," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 20, no. 5, pp. 818–826, 2012.

[20] "Nvidia professional graphics solutions." http://www.nvidia.com/content/PDF/line_card/5409_NV_ProGraphicsSolutions_LineCard_FEB13_HR.pdf. NVIDIA Corporation, Accessed: 2013-02-14.

[21] NVIDIA Corporation, *NVIDIA Nsight Development Platform, Visual Studio User Guide*, 2.2 ed., 2014. http://http.developer.nvidia.com/NsightVisualStudio/2.2/Documentation/UserGuide/HTML/Content/Timeout_Detection_Recovery.htm.

[22] A. Grama, *Introduction to Parallel Computing*. Pearson Education, Addison Wesley Publishing Company Incorporated, 2003.

[23] NVIDIA Corporation, *NVIDIA CUDA Library Documentation*, 4.1 ed., 2014. http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/index.html.

[24] NVIDIA Corporation, *NVIDIA CUDA Compiler Driver NVCC Documentation*, 2014. http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#further-mechanisms.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD–MM–YYYY)* 27–03–2014	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2012–Mar 2014

4. TITLE AND SUBTITLE Numerical Integration with Graphical Processing Unit for QKD Simulation	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Garrett, Virginia R., Captain, USAF	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, Graduate School of Engineering and Management (AFIT/EN), 2950 Hobson Way, WPAFB, OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-14-M-33
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Laboratory for Telecommunication Science, Attn: Gerry Baumgartner, 8080 Greenmead Dr., College Park, MD 20740, gbaumgartner@ltsnet.net	**10. SPONSOR/MONITOR'S ACRONYM(S)** LTS
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

The Air Force Institute of Techology (AFIT) is developing a simulation framework to model a wide variety of existing and proposed Quantum Key Distribution (QKD) systems. This research investigates using graphical processing unit (GPU) technology to more efficiently integrate optical pulses modeled within this framework. The goal is to reduce the simulation execution time. A GPU algorithm is presented for performing numerical integration of optical pulses described by Gaussian curves to improve pulse energy and power calculations. In order to measure the performance of the algorithm a optimal timing method is needed. A timer using Comute Unified Device Architecture (CUDA) events is selected over a Windows system application programming interface (API) timer. The problem sizes studied produce speedups greater than 60x on the NVIDIA Tesla C2075 compared to the Intel i7-3610QM CPU.

15. SUBJECT TERMS

Software Engineering, GPU Programming, Numerical Methods, Quantum Key Distribution

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Douglas Hodson, AFIT/ENG
U	U	U	UU	74	**19b. TELEPHONE NUMBER** *(include area code)* (937) 785-3636 x4719